Learning Score-Optimal Chordal Markov Networks via Branch and Bound

Kari Rantanen

MSc thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, September 27, 2017

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author

Kari Rantanen

Työn nimi — Arbetets titel — Title

Learning Score-Optimal Chordal Markov Networks via Branch and Bound

Oppiaine — Läroämne — Subject

Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	September 27, 2017	54

Tiivistelmä — Referat — Abstract

Graphical models are commonly used to encode conditional independence assumptions between random variables. Here we focus on undirected graphical models called *chordal Markov networks*. Specifically, we will consider the *chordal Markov network structure learning* problem (CMSL), where the aim is to find (or "learn") a graph structure that best fits the given data with respect to a given decomposable scoring function.

We introduce a branch and bound search algorithm for CMSL which represents chordal Markov network structures as decomposable DAGs. We show how revisiting equivalent solution candidates can be avoided in the search by detecting symmetries among graph structures. For the symmetry breaking we apply specific rules by van Beek and Hoffman (CP 2015), and also propose a new rule that takes advantage of the special nature of decomposable DAGs. In addition, we show how we can achieve on-the-fly score pruning for CMSL.

We also propose methods for obtaining strong upper bounds for CMSL that help us close branches in the search tree. We implement a dynamic programming algorithm to find the optimal *Bayesian* network structures and then use the scores of those graphs as upper bounds. We also show how we can relax the requirement for decomposability in decomposable DAGs in order to achieve even stronger upper bounds. Furthermore, we propose a method for obtaining an initial lower bound in CMSL by turning a Bayesian network structure into a chordal Markov network structure.

Empirically we show that our approach is competitive with the recently proposed CMSL algorithms by being able to sometimes scale up to 20 variables within 24 hours with unbounded treewidth. We also report that our branch and bound requires considerably less memory than the fastest of the recently proposed algorithms for CMSL.

Avainsanat — Nyckelord — Keywords

Chordal Markov Networks, Bayesian Networks, Exact Structure Learning, Branch and Bound

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction **1**

2 Preliminaries **5**
2.1 Bayesian networks 5
2.2 Chordal Markov networks 9
2.3 Clique trees 12
2.4 Dynamic programming 14
2.5 Exact algorithms for CMSL 16

3 Branch and bound for CMSL **18**
3.1 The branch and bound 18
3.2 Symmetry breaking 20
3.3 Dynamic choice construction 27
3.4 Bounds for CMSL 28
3.4.1 BNSL-based upper bounds 29
3.4.2 Improved upper bounds 31
3.4.3 Lower bounds 34
3.5 Example run of the algorithm 36

4 Empirical evaluation **40**
4.1 Experiment setup 40
4.2 Running time comparison 41
4.3 Memory usage and the impact of bounding 46

5 Conclusions and further work **49**

References **52**

1 Introduction

Many fields, such as machine learning, computational physics and cognitive science, deal with different kinds of data, and the ability to access the underlying probability distributions is often crucial [7]. However, with a large number of variables, storing these distributions can take a lot of space, and queries might be very inefficient to perform as well. The study of *Graphical models* aims at providing tools for tackling these problems [1, 39, 25].

The key idea behind graphical models is to capture the conditional independences of given random variables [23]. This is done by storing a graph structure where each vertex corresponds to one of the variables and the edges are used to encode the underlying dependencies. The graph can then be used to reduce the size of factors that we need to represent the entire joint probability distribution of the variables. This way we are often able to compress an exponentially-sized probability distribution into a tractable-sized representation [23]. The more accessible representation can also speed up certain queries on the distribution.

There are several methods for constructing a graphical model. One of them would be to apply formal knowledge, such as a system design, to synthesize a model. It is also possible to take a more subjective route, where a group of people uses their expertise to capture a situation into a graphical model [7]. In both cases a good prior knowledge is needed about the target to be modeled. However, here we focus on *data-driven* approaches, where the graphical model is automatically learned based on given data, such as medical or voting records [23]. Both the network structure (a graph encoding the conditional independence assumptions) and the parametrization (probabilities) can be learned from data. Here we focus exclusively on the structure learning side.

Specifically, we consider the *chordal Markov structure learning* problem (CMSL), where the aim is to find a specific type of undirected graphical model that best fits given data [8, 40, 26, 25]. In literature chordal Markov networks are sometimes referred as triangulated Markov random fields or decomposable models.

The CMSL algorithms considered here will be *score-based* approaches. In the score-based setting, we treat the structure learning task as an optimization problem, where we try to find a network structure that maximizes a given objective function [2]. The scoring function is constructed based on given data so that the solutions to the optimization problem will be best-fitting network structures for the data in terms of the scoring function. Scoring functions, such as the ones considered in this work, provide a compromise

between data fit and model complexity [23], and thereby aim at avoiding overfitting, a typical problem in machine learning.

Finding a maximum likelihood chordal Markov network with bounded structure complexity is known to be NP-hard [31]. For this reason several Markov chain Monte Carlo approaches have been proposed [27, 36, 16, 17]. CMSL is very closely related to another well-known problem, Bayesian network structure learning (BNSL) [23], but can appears to be computationally more challenging. For example, a recent exact approach to CMSL, based on constraint optimization, did not scale up to 10 variables [5]. A similar approach was also taken in [24] in the form of a direct integer programming encoding for CMSL, but was not empirically evaluated in an exact setting. GOBNILP, an efficient integer programming approach for BNSL, was expanded for CMSL [2, 32]; the implementation scales up to 15 variables within an hour. To the best of our knowledge, the currently fastest implemented algorithm for CMSL is Junctor, a dynamic programming approach based on recursive characterization of clique trees [20, 21]. In general, none of the recently proposed algorithms for CMSL seem to be able to solve a 20-variable instance within 24 hours, whereas any 20-variable BNSL instance can be solved within one minute with dynamic programming, as we demonstrate in Section 4.

In this thesis we introduce a new algorithm for learning a score-optimal chordal Markov networks. Although the ultimate output of our algorithm is an undirected graph, we will instead work with *decomposable* DAGs (directed acyclic graphs) which are an alternative representation form for chordal Markov network structures. In this respect our approach is connected to *CPBayes*, a recently proposed exact BNSL algorithm by van Beek and Hoffmann [38], which specifically operates on DAGs. The main similarity between our algorithm and CPBayes is that both of them utilize a branch and bound search for finding an optimal network structure. However, the branch and bounds are of very different nature, due to the different learning problems being solved. Because of the properties of BNSL, CPBayes merely searches for an optimal permutation of the variables, since the optimal graph structure for each permutation is trivial to determine [38]. This is not the case with CMSL, and as such merely searching over permutations does not suffice. Instead, here we have to consider a larger variety of options, and so we propose a dynamic programming algorithm that, within each search tree node, determines these necessary options.

Van Beek and Hoffman [38] introduced several powerful score pruning rules and symmetry-breaking constraints for BNSL, most of which are not applicable to CMSL. However, we are able to utilize their depth constraints to rule out some equivalent solutions from the search space. Futhermore, CPBayes implements symmetry breaking for BNSL based on so-called covered edges,

originally proposed by Chickering [3, 4], which is applicable to CMSL. Here we propose a new symmetry-breaking rule for CMSL, which covers the covered edge rules, but allows for considerable amount of additional pruning by taking advantage of the special properties of decomposable DAGs. We will also introduce a method for achieving on-the-fly score pruning for CMSL.

In order to efficiently close branches in the search tree, we propose ways to calculate strong upper bounds for CMSL. Since BNSL is currently much faster to solve than CMSL, one of our strategies is to compute the scores of optimal Bayesian network structures and then use those scores as upper bounds for CMSL. For this purpose we implement a dynamic programming algorithm by Silander and Myllymäki [30]. In addition, we show how to obtain even stronger upper bounds by relaxing the requirement of decomposability in the solutions. Finally, we propose a way for obtaining lower bounds for CMSL by taking an optimal Bayesian network structure and then turning it into a feasible decomposable DAG. Our empirical results suggest that the methods proposed in this thesis form a way of learning chordal Markov networks that is competitive with the recently proposed CMSL algorithms.

The structure of this thesis is as follows. Section 2 includes preliminary definitions and concepts that are necessary for understanding the rest of the thesis. Specifically, we will start by introducing Bayesian networks and the problem of learning them in Section 2.1. Then, in Section 2.2, we will proceed to introduce chordal Markov networks and the equivalent decomposable models, and the problem of learning these structures. Section 2.3 introduces clique trees, which are an alternative way for representing chordal Markov network structures, and also provide insight to some of the properties of decomposable models. In Section 2.4 we explain dynamic programming, the essential algorithm paradigm that is used in various subroutines in our branch and bound. We end the preliminaries in Section 2.5 by giving a brief overview of the different approaches that have been developed for chordal Markov network structure learning.

In Section 3 we introduce our approach for CMSL. First, we explain the core branch and bound in Section 3.1. In Sections 3.2 and 3.3, we explain how to reduce the search space with the use of symmetry-breaking constraints and on-the-fly score pruning. Section 3.4 shows to calculate strong bounds for CMSL. We end introducing our approach in Section 3.5 by providing an example of how our algorithm behaves with an example instance, tying together the concepts introduced in the preceding sections.

Section 4 contains the results of our empirical evaluation, benchmarking our approach under different settings, and comparing its performance to the fastest CMSL algorithms. Finally, we will give concluding remarks of our

findings in Section 5.

A paper covering the contributions of this thesis, titled "Learning Chordal Markov Networks via Branch and Bound", has been accepted for publication in the proceedings of the 30th Annual Conference on Neural Information Processing Systems (NIPS 2017).

2 Preliminaries

In this section we overview key concepts that will be used throughout the thesis. Although the main subject of this thesis concerns the learning of chordal Markov network structures, we will start by defining Bayesian networks (BN). This is because we will represent chordal Markov networks as structures that are actually a specialization of BN structures. Also, we will make use of BN structure learning in our approach to CMSL.

2.1 Bayesian networks

In the scope of this thesis we will assume that Bayesian networks (and chordal Markov networks) are constructed based on multivariate discrete data. Let $V = \{v_1, ..., v_n\}$ be a set of random variables and let $X_i = \{x_{i,1}, \ldots, x_{i,k_i}\}$ denote the set of possible values for each $v_i \in V$. We can represent the data as a $m \times n$ data matrix D, where m is the sample size of the data [30]. That is, each of the n columns in D represent one of the variables, and each row of the matrix is a *sample*; an n-vector assigning a value for each of the variables [30]. For example, $D_{i,j} = x$ means that the ith sample of D assigns value $x \in X_j$ for variable v_j.

The data can be synthetically generated or a collection of real-world observations. To give an concrete example, one of the data sets used in our empirical evaluation, *Voting*, includes votes of each of the U.S. house of representatives congressmen on different issues. Here each sample (row in the data matrix) represents one of the 435 congressmen, whereas each variable (column in the matrix) specifies an attribute attached to the congressman. The first variable specifies whether the person was a republican or democrat, and the following 16 variables state whether the person voted for or against a given issue. However, in the context of this work we consider structure learning which does not concern what the given dataset represents or what method was used to gather the samples.

We will use the following definition for Bayesian networks [7].

Definition 1. *Given a set of variables V, a Bayesian network (BN) is a pair $(G, \mathcal{P})$ where*
- $G = (V, E)$ is the network structure*; a directed acyclic graph over V, and*
- $\mathcal{P}$ is the network parametrization*; a set of factors representing the joint probability distribution of V.*

Since the network structure contains one vertex corresponding to each of the

variables of the network, we will be using the terms "variable" and "vertex" interchangeably. We now shortly detail some basic graph-specific definitions.

Let $G = (V, E)$ be a directed acyclic graph (DAG). We define the parent set of v in G as $pa_G(v) = \{v' : (v', v) \in E\}$. That is, a vertex $v_i \in V$ is a parent of $v_j \in V$ (v_j is the child of v_i) in G if the edge $v_i \to v_j$ exists. We call vertex *a source* if it has no parents. If the vertex is not included in any of the parent sets of other vertices, we call it *a sink*. Naturally, a vertex cannot contain itself in its parent set. Also the acyclity of DAGs restricts the parent set possibilities for vertices.

Example 1. *Figure 1 is an example of a DAG containing six vertices. Call this graph G. Now, for example, $pa_G(v_6) = \{v_1\}$, $pa_G(v_4) = \{v_2, v_3\}$ and $pa_G(v_1) = \emptyset$. The graph has one source node; v_1, and three sink nodes; v_5, v_6 and v_4.*

The network parametrization of a BN is a set of *conditional probability tables (CPT)*, each of which stores the conditional probability distribution of one of the variables [7]. The probability of any instantiation $v_1 = x_1, \ldots, v_n = x_n$ can be factorized as

$$\mathrm{P}(v_1 = x_1, \ldots, v_n = x_n) = \prod_{i=1}^{n} \mathrm{P}(v_i = x_i \mid v_\ell = x_\ell : v_\ell \in pa_G(v_i)),$$

where G is a Bayesian network structure containing vertices $v_1, \ldots, v_n$. This factorization encodes *conditional independence relations* according to d-separation property [23].

Therefore, given a Bayesian network $(G, \mathcal{P})$, each CPT for variable $v \in V$ only needs to store the probability distribution of v given all the possible instantiations of $pa_G(v)$ [7]. Source vertices are an exception since their parent sets are empty; their CPTs only need to store marginal probabilities of the corresponding variable getting its possible values.

Example 2. *Consider the CPT in Table 1. Here we see, for example, that $P(v_4 = true \mid v_2 = true, v_3 = false) = 0.9$: That is, the probability for*

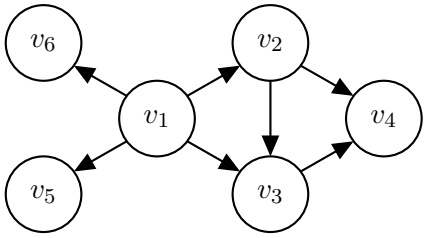

Figure 1: A directed acyclic graph; a possible Bayesian network structure.

$v_4 = true$ is 0.9 when we have $v_2 = true$ and $v_3 = false$. Note that v_2 must always be either true or false; for this reason we must have

$$P(v_4 = true \mid v_2 = true, v_3 = false) + P(v_4 = false \mid v_2 = true, v_3 = false) = 1.$$

This implies $P(v_4 = false \mid v_2 = true, v_3 = false) = 0.1$. Thus the probabilities for $v_4 = false$ can be inferred and are not needed in the CPT.

Given data D over variables V, and a network structure $G = (V, E)$,

$$\mathrm{P}(D \mid G) = \int_{\mathcal{P}} \mathrm{P}(D \mid \mathcal{P}, G) \cdot \mathrm{P}(\mathcal{P} \mid G)\ d\mathcal{P}$$

is the *marginal likelihood* of D given G [23], where $\mathrm{P}(D \mid \mathcal{P}, G)$ is the likelihood of the data given the Bayesian network $(G, \mathcal{P})$, and $\mathrm{P}(\mathcal{P} \mid G)$ is the prior distribution of G given parameters $\mathcal{P}$. In practice it is often necessary to use a logarithmic value $\log \mathrm{P}(D \mid G)$ of the marginal likelihood [5].

Now, when we want to find the network structure G that best fits the data D, we aim to maximize the *posterior probability* of G given D [23]:

$$\mathrm{P}(G \mid D) = \frac{\mathrm{P}(D \mid G) \cdot \mathrm{P}(G)}{\mathrm{P}(D)},$$

where $\mathrm{P}(D \mid G)$ is the marginal likelihood, $\mathrm{P}(G)$ is the prior probability of G and $\mathrm{P}(D)$ is a constant normalizing factor that can be ignored when solving the structure learning problem. The prior probabilities $\mathrm{P}(G)$ can be used to favor graphs that have a preferred structure. However, when using uniform priors the term $P(G)$ can be ignored, and thus the optimization problem reduces to maximizing the marginal likelihood [5]. That is, the aim is to find a network structure G that maximizes $\log \mathrm{P}(D \mid G)$.

Given a data matrix D over random variables V, $\log \mathrm{P}(D \mid G)$ gives rise to a decomposable scoring function that decomposes as a sum of local scores which assign a real number ("score") to each pair (v, S) where $v \in V$

v_2	v_3	v_4	$\mathrm{P}(v_4 \mid v_2, v_3)$
false	false	true	0.6
false	true	true	0.5
true	false	true	0.9
true	true	true	0.2

Table 1: An example CPT for variable v_4 in Figure 1, where the random values v_2, v_3 and v_4 take values from the set $\{true, false\}$.

and $S \subseteq V \setminus \{v\}$. That is, the scoring function assigns a value to each potential parent set choice that could occur in a BN structure. Concretely, the decomposability of the scoring function means that we can calculate the global score of a graph based on the parent sets of its vertices [2]. The scores are can be defined in such a way that finding a graph with the highest score equals to finding the network structure that maximizes the posterior probability [23]. There are several types of scoring functions to choose from, such as BDe(u), BIC and AIC [30]. However, we will not go into further details on how scoring functions are constructed, because we will assume in the context of this work that the scoring function is given to the structure learning algorithms as input.

Given a scoring function s over a set of variables V, we will use $s(v, S)$ to denote the score of variable $v \in V$ having $S \subseteq V \setminus \{v\}$ as its parent set. We are now ready to define the score of a Bayesian network structure.

Definition 2. *Let $V = \{v_1, ..., v_n\}$ be a set of variables and let s be a decomposable scoring function over V. Given a Bayesian network structure $G = (V, E)$, score of G is*

$$s(G) = \sum_{v \in V} s(v, pa_G(v)).$$

In words, the score of a Bayesian network structure is the sum of the scores of the parent set choices of the vertices in the DAG.

A *BNSL instance* is a pair (V, s), where $V = \{v_1, \ldots, v_n\}$ is a set of variables and s is a decomposable scoring function over V. In addition we may be given a parent set bound $b \in \mathbb{N}, 0 \leq b < n$. The bound constrains that each vertex in a feasible graph can have at most than b parents. That is, given $G = (V, E)$, we require $pa_G(v) \leq b$ for all $v \in V$. Intuitively, a small value for b results in sparser graphs and also in a smaller number of possible solution candidates for the instance.

Definition 3. *Given a BNSL instance $\mathcal{I} = (V, s)$, the Bayesian network structure learning (BNSL) problem is the task of finding a directed acyclic graph $G = (V, E)$ such that*

$$G = \arg\max_{G' \in \mathcal{G}} s(G'),$$

where $\mathcal{G}$ is the collection of all possible DAGs over V. Such a G is called an optimal solution *for $\mathcal{I}$.*

It should be noted that for many scoring functions there may exist several Bayesian network structures with equal score [30]. Thus it is sufficient to

find *any* network structure that maximizes the scoring function. For this reason we will mainly use the term *an* optimal solution. The same holds true for chordal Markov network structures.

We end this section by giving a brief overview of exact algorithms for BNSL.

Several branch and bound based approaches for BNSL have been proposed over the years. The first exact algorithm for BNSL was introduced by Suzuki [33] in 1996, which was a depth-first branch and bound that used Minimum Description Length as the scoring function. This work was extended with better bounding methods by Tian [37]. Depth-first branch and bounds were also proposed by Malone and Yuan [29] and van Beek and Hoffmann [38], latter of which used constraint programming to reduce the BNSL search space. Fan and Yuan [13] developed an improved method to calculate lower bounds in branch and bound based BNSL approaches, whereas Suzuki [34] showed how to achieve tighter upper bounds.

A dynamic programming based BNSL algorithm was developed by Silander and Myllymäki [30], inspired by an earlier approach by Koivisto and Sood [22]. Yuan and Malone [41] showed how Bayesian networks could be learned using an A*-like approach, where BNSL is viewed as the task of finding the shortest path in an implicit state-space search graph [41]. Jaakkola *et al.* [18] proposed using linear programming relaxations for learning optimal Bayesian network structures. A linear programming based approach was also taken later by Cussens and Bartlett [32], resulting in one of the fastest BNSL implementations at the moment [2].

2.2 Chordal Markov networks

In this section we will overview chordal Markov networks and the related optimization problem.

Let $G^u = (V, E)$ be an undirected graph. We say that G^u satisfies the global Markov property if and only if the following holds true: For each $S_a, S_b, S_c \subset V$, the sets of variables S_a and S_c are *conditionally independent* given S_b if and only if every path from a vertex in S_a to a vertex in S_c goes through a vertex in S_b [5]. For example, in Figure 2, $S_a = \{v_5, v_6\}$ is conditionally independent of $S_c = \{v_4\}$ given $S_b = \{v_2, v_3\}$.

A cycle has a *chord* if there is an edge between two non-consecutive vertices of the cycle [20]. Furthermore, an undirected graph is *chordal* if all its cycles with 4 or more vertices contain a chord.

Definition 4. *Given a set of variables V, a chordal Markov network is a*

pair $(G, \mathcal{P})$ *where*
- $G^u = (V, E)$ *is the network structure; a chordal undirected graph, and*
- $\mathcal{P}$ *is the network parametrization; a set of factors representing the joint probability distribution of* V*.*

We will now define decomposable DAGs, which are one way of representing chordal Markov network structures. Decomposable DAGs are very similar to Bayesian network structures, but use an additional concept of an *immorality* to restrict the set of possible graphs.

Definition 5. *Let* $G = (V, E)$ *be a DAG where* $(p, v) \in E$ *and* $(p', v) \in E$ *for* $v, p, p' \in V, p \neq p'$*. We say that* p *and* p' *are* an immorality *in* G *if and only if we have both* $(p, p') \notin E$ *and* $(p', p) \notin E$*.*

In words, a pair of vertices are an immorality in DAG if and only if (1) there is no edge between the vertices, and (2) the vertices share a common child vertex.

Definition 6. *Let* $G = (V, E)$ *be a DAG. If no pair of vertices* $v, v' \in V$ *causes an immorality in* G*, then we say that* G *is* a decomposable DAG*.*

Given any DAG G, we can construct its undirected version G^u by ignoring the edge directions. An undirected graph like this is called the *skeleton* of G. Now, if G is a decomposable DAG, then G^u is a chordal Markov network structure [8, 23]. Thus, it is possible to work with directed graphs when searching for optimal chordal Markov network structures. For example, GOBNILP uses this strategy for CMSL [2].

Decomposable scoring functions are constructed so that two DAGs have the same score if they share a same skeleton and contain no immoralities [32]. Hence such graphs are equivalent in terms of their score.

Definition 7. *Let* $G = (V, E)$ *and* $G' = (V, E')$ *be decomposable DAGs and let* G^u *and* $G^{u'}$ *be their skeletons, respectively. If* $G^u = G^{u'}$*, then* G *and* G' *are* equivalent*.*

A caveat of working with directed graphs is that for a chordal Markov network structure of n vertices, there can exist up to exponential number of corresponding decomposable DAGs with respect to the number of vertices. This happens, for example, when there is an edge between each pair of vertices (i.e. the graph is complete). For this reason the space of possible decomposable DAGs is considerably larger than the space of their skeletons. However, in Section 3.2 we introduce powerful ways of breaking these symmetries.

Example 3. *The graph in Figure 2 is the skeleton of the graph in Figure 1. Since the graph in Figure 1 is decomposable, this is a chordal Markov network structure. If we changed the edge $v_1 \to v_2$ into $v_1 \leftarrow v_2$ in Figure 1, we would still have a decomposable model matching the graph above. However, we cannot change the edge $v_2 \to v_4$ into $v_2 \leftarrow v_4$ in Figure 1 because then we would have an immorality between v_1 and v_4, and thus the result would no longer be a decomposable DAG.*

A CMSL *instance* is a pair (V, s), where $V = \{v_1, \ldots, v_n\}$ is a set of variables and s is a decomposable scoring function over V.

Definition 8. *Given a CMSL instance $\mathcal{I} = (V, s)$, The chordal Markov network structure learning problem is the task of finding a decomposable DAG G such that*

$$G = \arg\max_{G' \in \mathcal{G}} s(G'),$$

where $\mathcal{G}$ is the collection of all possible decomposable DAGs over V. G is called an optimal solution *to $\mathcal{I}$.*

As the set of possible decomposable DAGs is a subset of the set of possible DAGs (for a given number of variables), one might assume that CMSL would be computationally less challenging than BNSL. However, this is not the case; for instance, when the order of the vertices is fixed, the optimal Bayesian network structure (following that order) is unambiguous, and can be inferred in linear time with respect to the number of vertices [30, 38]. This is because greedily selecting the highest-scoring parent sets for each vertex (following the order) is enough; the order itself guarantees acyclity, which is the only requirement for feasibility in BNSL [18]. In CMSL, finding the optimal network is non-trivial even when the order of the vertices is fixed. This is because, due to the avoidance of immoralities, the set of possible parent set choices for a vertex heavily depends on the parent set choices that were made for the preceding vertices in the ordering.

Furthermore, approaches to BNSL often make use of score-pruning rules [11, 10] which, in contrast to CMSL, work under the assumption that immoralities

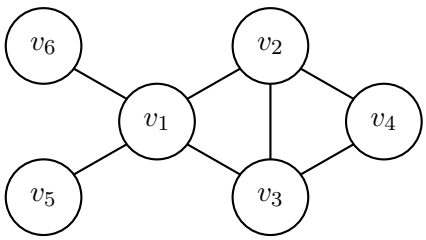

Figure 2: An undirected graph that is chordal.

are allowed. These rules typically allow for pruning out parent sets whose subsets satisfy a given criteria, so that the parent set could always be replaced by its subset [11, 10]. In Section 3.2 we will show that certain BNSL pruning rules are not applicable in the context of CMSL

2.3 Clique trees

In this section we will define an alternative representation for chordal Markov network structures based on clique trees [21]. This is because clique trees will be used in a proof later in the thesis, and because we will be comparing our CMSL algorithm to an algorithm that operates on clique trees.

We will start by providing the following definition for clique [20].

Definition 9. *Let (V, E) be an undirected graph and let $C \subseteq V$ be a subset of the vertices. If for all $v, v' \in C$, $v \neq v'$ we have $\{v, v'\} \in E$, then we call C a* clique*.*

In words, a clique is a set of vertices such that there is an edge between each pair of the vertices [20].

Definition 10. *Let $G = (V, E)$ be an undirected graph. A clique tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ over G is a tree where each node $C_i \in \mathcal{V}$ is a clique of G, and the following conditions hold true.*
I. $\bigcup_i C_i = V$,
II. if $\{v_\ell, v_k\} \in E$, then $\{v_\ell, v_k\} \subseteq C_i$ for some $C_i \in \mathcal{V}$, and
III. given $C_i, C_j, C_k \in \mathcal{V}$, if there is a path from C_i to C_k through C_j in $\mathcal{T}$, then we must have $C_i \cap C_k \subseteq C_j$. (The running intersection property)

Note that also decomposable DAGs represent chordal Markov network structures based on their cliques. Consider any decomposable DAG $G = (V, E)$ and each $v \in V$. We must have $pa_G(v) \setminus \{v'\} \subseteq pa_G(v')$ for some $v' \in pa_G(v)$. In words, the acyclity and the lack of immoralities require that each parent set contains a vertex that is a child vertex to all the other vertices in the parent set. This implies that each $\{v\} \cup pa_G(v)$ forms a clique in the skeleton of G.

In the discussion of Sections 2.1 and 2.2 we considered scoring functions that assigned scores to pairs of a variable and its possible parent set. However, we usually do not work in terms of parent sets when using the clique tree representation. We can calculate the score of a clique $C = \{v_1, \ldots, v_n\}$ by

$$s(C) = \sum_{i=1}^{n} s(v_i, \{v_{i+1}, \ldots, v_n\}),$$

which follows from the earlier observation about the relationship between cliques and parent sets in decomposable DAGs.

The intersections between the cliques of a clique tree are called *separators* [5]. Given set of cliques $\mathcal{C} = \{C_1, ..., C_n\}$ of a chordal Markov network structure G and the set of corresponding separators $\mathcal{S} = \{S_1, ..., S_{n-1}\}$, we obtain the following alternative way of calculating the score of G [20]:

$$s(G) = \sum_{C \in \mathcal{C}} s(C) - \sum_{S \in \mathcal{S}} s(S).$$

That is, the score of the graph is the product of the clique scores, subtracted by the product of the separators scores [8, 12]. Thus clique trees offer an alternative way to decomposable DAGs for representing chordal Markov network structures.

Sometimes we are given a CMSL instance with a treewidth bound. Clique trees allow us to define the treewidth of an undirected graph in the following way [23].

Definition 11. *Let $G^u = (V, E^u)$ be an undirected graph. Given all the possible clique trees $\mathcal{T}_1 = \{\mathcal{V}_1, \mathcal{E}_1\}, \ldots, \mathcal{T}_n = \{\mathcal{V}_n, \mathcal{E}_n\}$ over G^u, the* treewidth *of G^u is*

$$TW(G^u) = \min_{i=1,...,n} \max_{C \in \mathcal{T}_i} |C| - 1.$$

In words, the treewidth of an undirected graph G^u is the minimum *width* over all the possible clique trees over G^u. The width of a clique tree is the size of its largest clique minus 1. Bounding the treewidth of a CMSL instance means that a solution graph is considered feasible only if its treewidth does not exceed a given constant [20]. This is achieved by restricting the maximum parent set size.

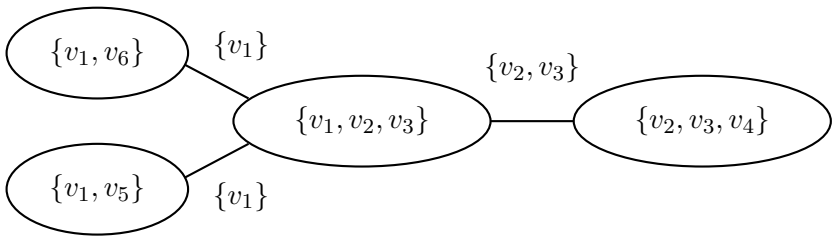

Figure 3: A clique tree representation of the undirected graph in Figure 2. The node labels represent the cliques, and the edge labels represent the separators between the cliques..

2.4 Dynamic programming

In this section we explain briefly the well-known generic algorithmic technique of dynamic programming [6], which is used heavily in various sub-procedures in the CMSL approach presented in this work.

Dynamic programming (DP) is a method of utilizing a *divide and conquer* strategy on a given problem and storing in memory (caching) already visited partial solutions [6]. In divide and conquer, we break the problem at hand into smaller and smaller subproblems, until we reach problems (called *base cases*) that are trivial to solve. The main challenge is to construct solutions to larger subproblems based on the solutions to smaller subproblems [6]. Using this strategy, we will finally obtain the solution to the original (full) problem.

Each time we solve a subproblem, we store its solution in a data structure called *DP table* as a pair of input and solution. The idea is to be able to efficiently retrieve any previously-computed solution using the input as the key [6]. An efficient way of implementing a DP table is to use a simple array that provides unit-time accesses and updates. However, this requires that all the input values have distinguished non-negative integer equivalences (i.e. can be directly used as unique array indexes). Also, the smallest and largest of these integers must be within a reasonable distance from each other; otherwise we might need to allocate an array that exceeds our memory capacity. One alternative would be to use a hash map, which could potentially provide an *amortized* unit-time accesses and updates.

Caching is an essential part of dynamic programming since it helps us avoid solving the same subproblems multiple times. In fact, ideally we need to solve each subproblem only once. For more complex problems, it is also possible to define multiple types of subproblems and therefore multiple DP-tables [20, 21].

We will now address how to efficiently cache subsets, since this happens very often with DP algorithms that work on graphs. Let $V = \{v_1, ..., v_n\}$ be the set of all vertices of a given BNSL/CMSL instance and let $I_W(v_i \in V)$ be an indicator variable for $W \subseteq V$ where $I_W(v_i) = 1$ iff $v_i \in W$ and $I_W(v_i) = 0$ iff $v_i \notin W$. Now, we can represent any $W \subseteq V$ with a non-negative integer $\sum_{i=1}^{n} 2^{i-1} \cdot I_W(v_i)$. These numbers are unique, and as such they can be used as array indexes. This requires $\Theta(2^n)$ memory for the DP table.

For the sake of simplicity, the pseudocodes of the DP algorithms that we will introduce in the coming sections will not include the caching part. For

this reason we will now represent a straight-forward method for adding the caching to those algorithms.

Let $I_1, ..., I_n$ and $O_1, ..., O_m$ be abstract sets, and let $f : \mathcal{I} \to \mathcal{O}$ be a function such that $\mathcal{I} = \{(i_1, ..., i_n) : i_1 \in I_1, ..., i_n \in I_n\}$ is the set of possible inputs for f and $\mathcal{O} = \{(o_1, ..., o_m) : o_1 \in O_1, ..., o_m \in O_m\}$ is the set of possible outputs for f. Now, we will construct g; a cached version of f, in the following fashion.

1. Initialize an empty DP-table $\mathcal{D}$ which is capable of storing the values from $\{(i, o) : i \in \mathcal{I}, o \in \mathcal{O}\}$. We use $\mathcal{D}[i] = o$ to denote the values in $\mathcal{D}$ for $i \in \mathcal{I}, o \in \mathcal{O}$.
2. Construct function $g : \mathcal{I} \to \mathcal{O}$ as follows. In the beginning of $g(i \in \mathcal{I})$ we will check if the value $\mathcal{D}[i]$ exists, and if so, let $g(i) = \mathcal{D}[i]$. Otherwise, we compute the solution $o \in \mathcal{O}$ like in f, but by replacing all the calls to f in the recursion with calls to g. Finally, we let $g(i) = D[i] = o$.

Example 4. *Let* FIBONACCI $: \mathbb{N} \to \mathbb{N}$ *be the following a algorithm to calculate Fibonacci numbers.*

```
1: function FIBONACCI(n)
2:     if n ≤ 1 then return n
3:     return FIBONACCI(n − 1) + FIBONACCI(n − 2)
4: end function
```

In this classic example we can see that the problem is being divided into smaller subproblems, and the solutions from those are then added up get a solution to a larger subproblem. The base case is at $n \leq 1$. As for the time complexity, the recursion gets constantly divided into two heavily-overlapping branches, which implies $\mathcal{O}(2^n)$.

Now, given a DP-table $\mathcal{D}$, we can construct the following DP algorithm DP-FIBONACCI*:*

```
1: function DP-FIBONACCI(n)
2:     if D[n] does not exist yet then
3:         if n ≤ 1 then return n
4:         D[n] ← DP-FIBONACCI(n − 1) + DP-FIBONACCI(n − 2)
5:     end if
6:     return D[n]
7: end function
```

Here the time complexity is $\mathcal{O}(n)$ since each value DP-FIBONACCI(0) ... DP-FIBONACCI(n) *is computed only once.*

2.5 Exact algorithms for CMSL

In this section we give a brief overview of algorithms that have been developed for score-optimal chordal Markov network structure learning.

Solving CMSL instances is possible with GOBNILP, one of the top approaches for Bayesian network structure learning [32]. GOBNILP implements a branch-and-cut approach where BNSL is modeled as a series of relaxed integer programs that are then solved with the SCIP optimization framework [2]. The relaxations are based on the observation that acyclity is the reason why BNSL is computationally hard, thus by allowing cycles one can get quickly (mostly infeasible) solutions to the problem [2]. These solutions are then used to guide the search by gradually deriving new constraints (cutting planes) until the optimal network structure is found. The program has an option for disallowing immoralities from the solutions, thus providing a way of learning network structures that are equivalent to chordal Markov networks [2]. Based on our empirical evaluation (Section 4), GOBNILP scales up to 15 variables within an hour with unbounded treewidth.

Corander *et al.* [5] and Janhunen *et al.* [19] developed a different approach for CMSL, based on creating a set of logical constraints to convey the properties of a desired solution. These constraints are then given to an external solver which eventually terminates with the optimal solution to the learning problem. The authors experimented with multiple ways to represent the constraints, including maximum satisfiability, satisfiability modulo theories and answer-set programming [5].

To obtain efficient encodings, the following balancing condition was introduced: A clique graph is balanced if for all vertices the number of cliques containing the vertex is one higher than the number of edges attached to that vertex [5]. Since all the balanced spanning trees of a clique graph have the same score, the balancing constraint is used to prune considerable amount of equivalent solutions from the search space [5]. This strategy allowed their approach to scale up to 8 variables with unbounded treewidth [19].

A similar approach to that of Corander *et al.* [5] was also provided by Kumar and Bach [24] in the form of direct integer programming, but the method was not empirically evaluated in an exact setting.

Most recently, Kangas *et al.* [20, 21] developed a dynamic programming

algorithm *Junctor* for learning chordal Markov network structures. Junctor divides CMSL into the following three types of subproblems [20].

1. Given a separator S in a clique tree and a set of unassigned vertices U, find a clique $S \subset C \subseteq S \cup U$ that, if added to the tree, would lead to a locally optimal solution.
2. Given a clique C in a clique tree and a set of unassigned vertices U, find a way to partition U into subsets $R_1, ..., R_k \subseteq U$ so that each subset corresponds to a separate branch in the tree and that the partitioning would lead to a locally optimal solution.
3. Given a clique C in a clique tree and a set of unassigned vertices R, find a separator $S \subset C$ that, if was used by the tree, would lead to a locally optimal solution.

The three types of subproblems are able to aid each other: We can solve problem 1 by using the solutions to problem 2, solve 2 with solutions to 3 and solve 3 with solutions to 1. This leads to a recursive characterization of clique trees, where the globally optimal solution is found by solving problem 1 with $S = \emptyset$ and U being the set of variables of the CMSL instance [20]. Furthermore, Junctor uses three memoization tables, one for each subproblem, to cache the scores of already-visited partial solutions [20]. Hence, even thought Junctor has to iterate through all the possible solution candidates, the caching helps to avoid computing same tree structures multiple times. The program is also likely to benefit from its relatively simple nature which provides it a minimal performance overhead.

Junctor has $\Omega(4^n)$ time complexity and $\Omega(3^n)$ space complexity, where n is the number of variables [20]. Due to the dynamic programming, the algorithm offers extremely consistent running times; in our empirical tests it is able to solve any 17-variable instance within an hour and any 19-variable instance within 24 hours. It is one of the currently fastest practical algorithms for CMSL when the treewidth of the instances is not bounded [20]. On the other hand, when a given treewidth bound is low enough, Junctor is often outperformed by GOBNILP [20].

3 Branch and bound for CMSL

In this section we present details on our branch and bound approach to CMSL. We start with an overview of the search algorithm, and then explain how we apply symmetry breaking, make use of dynamic programming for computing parent set choices during search, and obtain strong bounds for pruning the search tree.

3.1 The branch and bound

In general it is possible to gradually construct a same DAG in multiple different ways by adding the vertices and edges to it in different order. Clearly, the search would be unnecessarily inefficient if identical solution candidates would be repeatedly considered. Thus we will define *ordered decomposable DAGs* which encode the precise order in which a graph was constructed during the search. This information can then be used to prune the search space by identifying symmetries (see Section 3.2).

Definition 12. *$G = (V, E, \pi)$ is an ordered decomposable DAG over variables $V = \{v_1, \ldots, v_n\}$ if and only if (V, E) is a decomposable DAG and $\pi : \{1...n\} \to \{1...n\}$ a total order over V such that $(v_i, v_j) \in E$ only if $\pi^{-1}(i) < \pi^{-1}(j)$ for all $v_i, v_j \in V$.*

In words, ordered decomposable DAGs forbid any edge $v_i \to v_j$ where v_j precedes v_i in the ordering.

Example 5. *Let $G = (V, E, \pi)$ be the ordered decomposable DAG represented in Figure 4. Because the graph contains the edge $v_1 \to v_3$, we must have $\pi^{-1}(1) < \pi^{-1}(3)$ (by Definition 12). Similarly we must have $\pi^{-1}(1) < \pi^{-1}(4)$, $\pi^{-1}(4) < \pi^{-1}(5)$ and $\pi^{-1}(4) < \pi^{-1}(2)$. Thus one of the possible orderings π for G is $v_1 \prec v_4 \prec v_5 \prec v_3 \prec v_2$.*

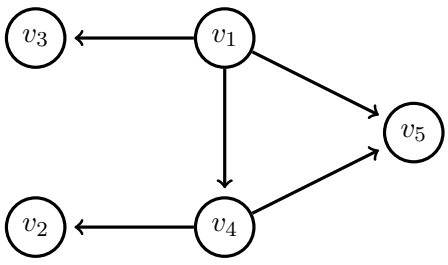

Figure 4: A decomposable DAG where the order of the vertices can be ambiguous.

Partial solutions during search are hence ordered decomposable DAGs, which are extended by adding a parent set choice (v_k, P), i.e., adding the new node v_k and edges from each of its parents in P (already in the DAG) to v_k.

Definition 13. *Let $U = \{v_1, \ldots, v_n\}$ be a set of variables, and let $G = (V, E, \pi)$ be an ordered decomposable DAG such that $V \cap U = \emptyset$. Given $v_k \in U$ and $P \subseteq V$, we say that the ordered decomposable DAG $G' = (V', E', \pi')$ is G with the parent set choice (v_k, P) if the following conditions hold.*

1. *$V' = V \cup \{v_k\}$.*
2. *$E' = E \cup \bigcup_{v' \in P} \{(v', v_k)\}$.*
3. *We have $\pi'(i) = \pi(i)$ for all $i = 1...|V|$, and $\pi'(|V| + 1) = k$.*

Algorithm 1 represents the core functionality of the branch and bound. The recursive function takes two arguments; the remaining vertices of the problem instance, U, and the current partial solution $G = (V, E, \pi)$. In addition we keep stored a best lower bound solution G^*, which is the highest-scoring solution that has been found so far. Thus, at the end of the search, G^* is an optimal solution. During the search we use G^* for bounding as further detailed in Section 3.4.2.

In the loop on line 7 we branch with all the necessary parent set choices for the current partial solution: The procedure ParentSetChoices(U, G) and the related symmetry breaking are explained in Sections 3.2 and 3.3. We sort the parent set choices into decreasing order based on their score, so that (v, P) is tried before (v', P') if $s(v, P) > s(v', P')$, where $v, v' \in U$ and $P, P' \subseteq V$. This is done to focus the search first to the most promising branches for finding an optimal solution. When $U = \emptyset$, we have ParentSetChoices$(U, G) = \emptyset$, and so the current branch gets terminated.

Choosing the parent set becomes trivial when U contains only one vertex. In this case future immoralities are not a concern, and thus it is sufficient to greedily select the highest-scoring available parent set for the final vertex. When the parent sets for each vertex are stored in segment trees based on their score, it is possible to choose the last parent set in $\mathcal{O}(n)$ time, where n is the number of vertices of the problem instance. This is achieved by checking the highest-scoring subset within each of the cliques in the partial solution, very similarly to the technique used later in Section 3.4.1. This is faster than iterating through all the available parent set choices for the last vertex, which requires up to $\mathcal{O}(2^{n-1})$ time.

Algorithm 1 The core branch and bound search.

```
1: function BranchAndBound(U, G = (V, E, π))
2:     if U = ∅ and s(G*) < s(G) then
3:         G* ← G                          ▷ Update the lower bound if improved.
4:     end if
5:     if this branch cannot improve LB then return          ▷ Backtrack
6:     for (v_i, P) ∈ ParentSetChoices(U, G) do    ▷ Iterate the current
7:                                                    parent set choices.
8:         Let G' = (V', E', π') be G with the parent set choice (v_i, P).
9:         BranchAndBound(U \ {v_i}, G')          ▷ Continue the search.
10:    end for
11: end function
```

3.2 Symmetry breaking

We continue by proposing symmetry breaking for the space of ordered decomposable DAGs.

Similarly as van Beek and Hoffmann [38] for BNSL, we define the depths of vertices as follows.

Definition 14. *Let $G = (V, E, \pi)$ be an ordered decomposable DAG. The* depth *of $v \in V$ in G is*

$$d(G, v) = \begin{cases} 0 & \text{if } pa_G(v) = \emptyset, \\ \max\limits_{v' \in pa_G(v)} d(G, v') + 1 & \text{otherwise.} \end{cases}$$

Definition 15. *The depths of G are* ordered*, if for all $v_i, v_j \in V$ where $\pi^{-1}(i) < \pi^{-1}(j)$, the following conditions hold.*

1. *$d(G, v_i) \leq d(G, v_j)$.*
2. *If $d(G, v_i) = d(G, v_j)$, then $i < j$.*

In words, condition 1 of Definition 15 states that vertices, which come earlier in the ordering, cannot have a larger depth than the vertices that come later in the ordering. That is, the depths are monotonically increasing with respect to the ordering function. Condition 2 states that if two vertices have the same depth, they must be ordered lexicographically.

Example 6. *Let $G = (V, E, \pi)$ be the ordered decomposable DAG represented in Figure 4. We have $d(G, v_1) = 0$, $d(G, v_4) = 1, d(G, v_3) = 1, d(G, v_2) = 2$ and $d(G, v_5) = 2$. When we require the depths of G to be ordered (Definition 15), we can infer the exact order of the vertices as follows.*

Condition 1 implies that $\pi^{-1}(1) = 1$, $\pi^{-1}(4) < \pi^{-1}(5)$, $\pi^{-1}(4) < \pi^{-1}(2)$, $\pi^{-1}(3) < \pi^{-1}(5)$ *and* $\pi^{-1}(3) < \pi^{-1}(2)$. *Condition 2 implies that* $\pi^{-1}(3) < \pi^{-1}(4)$ *and* $\pi^{-1}(2) < \pi^{-1}(5)$. *Therefore the order of the vertices must be* $v_1 \prec v_3 \prec v_4 \prec v_2 \prec v_5$.

Chickering [3, 4] proposed using so-called covered edges to detect equivalences between Bayesian network structures. This technique was implemented in the form of symmetry-breaking constraints by van Beek and Hoffmann [38] to reduce the search space in their BNSL approach. Here we will introduce a concept of *preferred vertex order*, which is a generalization of covered edges, and only works for decomposable DAGs.

Definition 16. *Let* $G = (V, E, \pi)$ *be an ordered decomposable DAG. A pair* $v_i, v_j \in V$ *violates the preferred vertex order in* G *if the following conditions hold.*

1. $i > j$.
2. $pa_G(v_i) \subseteq pa_G(v_j)$.
3. *There is a path from* v_i *to* v_j *in* G.

Example 7. *Consider Figure 5. In the graph (a) we have* $pa_G(v_3) \subseteq pa_G(v_1)$ *and* $pa_G(v_3) \subseteq pa_G(v_4)$, *and there are paths from* v_3 *to both* v_1 *and* v_2. *Thus the pairs* (v_1, v_3) *and* (v_2, v_3) *violate the preferred vertex order, and graph (a) would get pruned.*

In graph (b) we see that the vertices v_3 *and* v_2 *are the only case where there is a directed path to a lexicographically smaller vertex (Conditions 1 and 3 of Definition 16). However, as we have* $pa_G(v_3) = \{v_1\} \not\subseteq \{v_3\} = pa_G(v_2)$, *the pair* (v_2, v_3) *does not violate the preferred vertex order. Thus the graph (b) would not get pruned.*

The graphs are equivalent, because they share the same skeleton and neither contains immoralities. In this case the only difference between the graphs

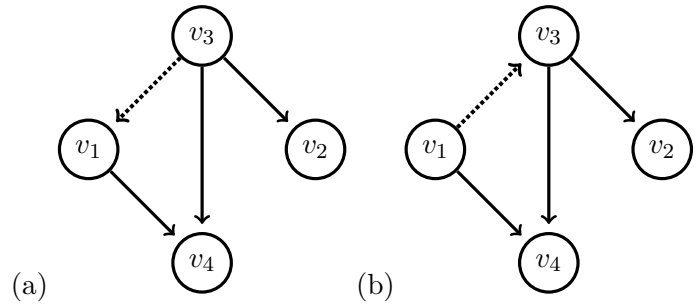

Figure 5: An example of fixing a violation of preferred vertex order in a DAG by flipping an edge.

is the direction of the edge between v_1 and v_3. In general equivalent graphs can differ by multiple edges, resulting in exponential number of equivalent graphs with respect to the number of vertices. This highlights the importance of breaking symmetries.

Next we will show that it is justified to use the concept of preferred vertex order to prune the search space without dismissing an optimal solution along the way. We start by defining a way to (lexicographically) compare total orders.

Definition 17. *Let $V = \{v_1, ..., v_n\}$ be a set of vertices and let π and π' be total orders over V. Let $k = min\{i \mid \pi(i) \neq \pi'(i)\}$ be the first difference between the orders. We denote $\pi = \pi'$ if no such difference exists. Otherwise $\pi < \pi'$ if and only if $\pi(k) < \pi'(k)$.*

We will now introduce a lemma that is used in the proof. The following lemma establishes that if a violation of the preferred order exists, we are able to fix it so that the lexicographical value of the resulting graph increases in the process.

Lemma 1. *Let $G = (V, E, \pi)$ be an ordered decomposable DAG. If there exists any $v_i, v_j \in V$ such that the pair (v_i, v_j) violates the preferred vertex order in G, then there exists an ordered decomposable DAG $G' = (V, E', \pi')$ where*

1. *G' belongs to the same equivalence class with G,*
2. *the pair (v_i, v_j) does not violate the preferred vertex order in G', and*
3. *$\pi < \pi'$.*

Proof. We begin by defining a directed clique tree $\mathcal{C} = (\mathcal{V}, \mathcal{E})$ over G.

Given $v_k \in V$, let $C_k = pa_G(v_k) \cup \{v_k\}$ be the clique defined by v_k in G. The nodes of $\mathcal{C}$ are these cliques; we also add an empty set as a clique to make sure the cliques form a tree (and not a forest). Formally, $\mathcal{V} = \{C_k : v_k \in V\} \cup \{\emptyset\}$.

Given $v_k \in V$ where $pa_G(v_k) \neq \emptyset$, let $\phi_k = \text{argmax}_{v_\ell \in pa_G(v_k)} \pi^{-1}(\ell)$ denote the parent of v_k in G that is in the least significant position in π. Now, the edges of $\mathcal{C}$ are

$$\begin{aligned}\mathcal{E} = &\{(\emptyset, C_k) : C_k = \{v_k\}, v_k \in V\} \cup \\ &\{(C_\ell, C_k\} : v_\ell = \phi_k, C_k \neq \{v_k\}, v_k \in V\}.\end{aligned}$$

In words, if $v_k \in V$ is a source vertex in G (i.e. $C_k = \{v_k\}$), then the parent of C_k is $\emptyset$ in $\mathcal{C}$. Otherwise (i.e $C_k \neq \{v_k\}$) the parent of C_k is C_ℓ, where v_ℓ

is the closest vertex to v_k in order π that satisfies $C_\ell \cap pa_G(v_k) \neq \emptyset$. We see that all the requirements for clique trees (Definition 10) hold for $\mathcal{C}$:

(I) $\bigcup_{C \in \mathcal{V}} C = V$.

(II) If $\{v_\ell, v_k\} \in E$, then either $\{v_\ell, v_k\} \subseteq C_k$ or $\{v_\ell, v_k\} \subseteq C_\ell$.

(III) Due to the decomposability of G, we have $C_a \cap C_c \subseteq C_b$ on any path from C_a to C_c through C_b (the running intersection property).

Now, assume that there exists $v_i, v_j \in V$ such that the pair (v_i, v_j) violates the preferred vertex order in G; that is, we have $i > j$, $pa_G(v_i) \subseteq pa_G(v_j)$ and a path from v_i to v_j in G. This means that there is a path from C_i to C_j in $\mathcal{C}$ as well.

Let $P \in \mathcal{V}$ be the parent node of C_i in $\mathcal{C}$. We see that C_j exists in a subtree $\mathcal{T}$ of $\mathcal{C}$, which is separated from rest of $\mathcal{C}$ by P, and where C_i is the root node. Let $\mathcal{T}'$ be a new clique tree, which is like $\mathcal{T}$, but redirected so that C_j is the root node of $\mathcal{T}'$. Let $\mathcal{C}'$ be a new clique tree, which is like $\mathcal{C}$, but $\mathcal{T}$ is replaced with $\mathcal{T}'$.

We show that $\mathcal{C}'$ is a valid clique tree. First of all, the nodes (cliques) of $\mathcal{C}'$ are exactly the same as in $\mathcal{C}$, so $\mathcal{C}'$ clearly satisfies the requirements I and II. As for the requirement III, consider the non-trivial case where $C_a, C_b \in \mathcal{C}$ have a path from C_a to C_b through C_i in $\mathcal{C}$. This means $v_i \notin C_a$ (due to the way $\mathcal{C}$ was constructed), and so we get

$$\begin{aligned} C_a \cap C_b \subseteq C_i \quad &\rightarrow C_a \cap C_b \subseteq C_i \setminus \{v_i\} \\ \rightarrow C_a \cap C_b \subseteq pa_G(v_i) \quad &\underset{\text{Def. 3 (2)}}{\subseteq} pa_G(v_j) \subseteq C_j. \end{aligned}$$

Therefore the running intersection property holds for $\mathcal{C}'$.

Let $\hat{\pi}$ be the total order by which $\mathcal{C}'$ is ordered. Let $G' = (V, E', \hat{\pi})$ be a new ordered decomposable DAG that is equivalent to G, but where the edges E' are arranged to follow the order $\hat{\pi}$.

Finally, we see that G' satisfies the conditions of the theorem:

1. The cliques of G' are identical to that of G, so G' belongs to the same equivalence class with G.

2. We have $\hat{\pi}^{-1}(j) < \hat{\pi}^{-1}(i)$, and therefore there is no path from v_i to v_j in G'. Thus the pair (v_i, v_j) does not violate the preferred vertex order in G'.

3. Let $o = \pi^{-1}(i)$. We have $\hat{\pi}(o) = j < i = \pi(o)$. Furthermore, the change from $\mathcal{T}$ to $\mathcal{T}'$ in $\mathcal{C}'$ did not affect any vertex whose position was earlier than o. Therefore $\hat{\pi}(k) = \pi(k)$ for all $k = 1...(o-1)$. This implies $\hat{\pi} < \pi$.

□

We are now ready to introduce the following theorem that justifies the use of preferred vertex order to prune the search space. We show here that even if fixing one violation introduces new violations, only a finite number of steps are needed to obtain a graph with no violations, since each step can be performed so that the lexicographical value of the solution increases (by Lemma 1).

Theorem 1. *Let $G = (V, E, \pi)$ be an ordered decomposable DAG. There exists an ordered decomposable DAG $G' = (V, E', \pi')$ that is equivalent to G, but where for all $v_i, v_j \in V$ the pair (v_i, v_j) does not violate the preferred vertex order in G'.*

Proof. Consider the following procedure for finding G':

1. Select $v_i, v_j \in V$ where the pair (v_i, v_j) violates the preferred vertex order in G. If there are no such vertices, assign $G' \leftarrow G$ and terminate.

2. Let π be the total order of the vertices of G. Construct an ordered decomposable DAG $\hat{G} = (V, \hat{E}, \pi')$, such that

   (I) the pair (v_i, v_j) does not violate the preferred vertex order in $\hat{G}$,
   (II) $\hat{G}$ belongs to the same equivalent class with G and
   (III) $\pi' < \pi$.

   Lemma 1 proves that $\hat{G}$ can be constructed from G.

3. Assign $G \leftarrow \hat{G}$ and return to step 1.

It is clear that when the procedure terminates, G' belongs to same equivalence class with G and there are no violations of the preferred vertex order in G'. We also see that the total order of G (i.e. π) is lexicographically strictly decreasing every time the step 3 is reached. There are finite amount of possible permutations (total orders) and therefore the procedure converges. The existence of this procedure and its correctness proves that G' exists. □

Mapping to practice, Theorem 1 allows for very effectively pruning out all symmetric solutions but the one not violating the preferred vertex order within our branch and bound approach.

The concept of preferred vertex order implicitly forces the lexicographically smallest vertex to be a source vertex in the graph that contains it. This can be seen as follows. Consider any nonempty directed graph $G = (V, E)$ where the lexicographically smallest vertex is $v_1 \in V$. Now, if $v_1 \in V$ is not a source, then v_1 exists in a subgraph of G where a vertex $v_i \in V$ is the source. Clearly this means that

1. $i > 1$,
2. $pa_G(v_i) = \emptyset \subseteq pa_G(v_1)$ and
3. there is a path from v_i to v_1.

Thus the pair (v_i, v_1) violates the preferred vertex order in G.

Note that "violates the preferred vertex order" concerns the order in which the vertices are in the underlying DAG, whereas "depths are ordered" concerns the order by which a solution was constructed. We use the former to prune whole solution candidates from the search space, and the latter to ensure that no solution candidate is seen twice during search.

Intuitively, the smaller the total order of the partial solution is (Definition 17), the less likely is adding a new vertex to the solution to cause a violation of the preferred vertex order. For example, if a solution is constructed during the search by adding the vertices in lexicographic order $(v_1, v_2, \ldots)$, then the first and third condition of Definition 16 can never hold at the same time. This gives rise to the following important observation.

Let $V = \{v_1, v_2, ..., v_n\}$ be the vertices of a CMSL problem instance. When we require the depths to be ordered and that there are no violations of the preferred vertex order, it becomes particularly important to fix the lexicographically smallest vertex $v_1 \in V$ as the first vertex in the order. Otherwise, if some other vertex is chosen to be the first, then we would enter a branch in the search that does not contain any valid solutions with respect to the symmetry-breaking constraints. This is because v_1 needs to be a source vertex (as explained earlier), but it could no longer be assigned on depth 0 because the depth would already contain a lexicographically larger vertex.

So far we have shown how pruning can be achieved for CMSL. Now, on the contrary, we will show that certain BNSL pruning rules are not applicable to CMSL. The rules are by van Beek and Hoffman [38] and use the notation $domain(v)$ to denote the set of possible parent sets for a vertex v. In addition, a parent set $P \in domain(v)$ is *consistent* for vertex v in a decomposable DAG G if having P as the parent set of v would preserve the acyclity and the lack of immoralities in G.

Theorem 2. *(Refutation to Theorem 3 from [38] in the case of CMSL.) Let $G = (V, E)$ be a decomposable DAG, and let $P, P' \in domain(v), P \neq P'$ be possible parent sets for $v \in V$. Even if $s(P) \geq s(P')$ and P is consistent for v in G, it is still possible that we cannot safely prune P' from $domain(v)$ in CMSL.*

Proof. Let $G = (V, E)$ be a decomposable DAG where $V = \{v_1, v_2, v_3\}$ and $E = \emptyset$, and let the BDeu scores be the following (given as pairs of a parentset and its score):

$$\begin{aligned} domain(v_1) = \{&(\emptyset, -152.04124751), (\{v_2\}, -152.10200712), \\ &(\{v_3\}, -154.71116923), (\{v_2, v_3\}, -4.90692972)\}, \\ domain(v_2) = \{&(\emptyset, -151.88351388), (\{v_1\}, -151.94427349), \\ &(\{v_3\}, -154.71116923), (\{v_1, v_3\}, -4.90692972)\}, \\ domain(v_3) = \{&(\emptyset, -149.27435178), (\{v_1\}, -151.94427349), \\ &(\{v_2\}, -152.10200712), (\{v_1, v_2\}, -4.90692972)\}. \end{aligned}$$

Clearly $\emptyset$ is consistent for each $v_i \in V$ in G. We also have $s(\emptyset) \geq s(\{v_j\})$ for any $v_j \neq v_i$ in $domain(v_i)$. However, by pruning those parentsets we could no longer construct a graph where the vertices form a clique. The clique would provide the globally optimal score of $s(v_1, \emptyset) + s(v_2, \{v_1\}) + s(v_3, \{v_1, v_2\}) = -152.04124751 + (-151.94427349) + (-4.90692972) = -308.89245072$. □

Theorem 3. *(Refutation to Theorems 4 and 5 from [38] in the case of CMSL.) Given a vertex v and its possible parent sets $P, P' \in domain(v)$, even if $s(P) \geq s(P')$ and $P \subset P'$, it is still possible that P' cannot be safely pruned from $domain(v_i)$ in CMSL.*

Proof. Let $G = (V, E)$ be any decomposable DAG containing vertices v and v', and let $P \subset P'$ for some $P, P' \in domain(v)$. It is clear that $v' \in P$ implies $v' \in P'$. Thus, if adding the edge $v' \to v$ would form a cycle or an immorality in G, then $v' \in P$ would imply that neither P or P' are consistent for v in G. In other words, P' is consistent for v in G only if P is consistent for v in G. Therefore Theorem 2 completes the proof. □

Score-pruning is not the only aspect of BNSL techniques that is not fully applicable to CMSL. We will now provide an example of symmetry breaking for BNSL that does not work when immoralities are disallowed.

Theorem 4. *(Refutation to symmetry-breaking constraint 8 from [38] in the case of CMSL.) Given vertices v_i and v_j, even if $domain(v_i)$ would equal to $domain(v_j)$ when all the occurrences of v_j are replaced by v_i, it may not be sufficient to only apply a single ordering between v_i and v_j in CMSL.*

Proof. Let $V = \{v_1, v_2, v_3\}$ be the set of vertices and consider the following undirected graphs $G_1 = (V, E_1)$, $G_2 = (V, E_2)$ and $G_3 = (V, E_3)$: In G_1 we have $\{v_1, v_2\} \in E_1$ and $\{v_2, v_3\} \in E_1$. In G_2 we have $\{v_1, v_3\} \in E_2$ and $\{v_3, v_2\} \in E_2$. In G_3 we have $\{v_3, v_1\} \in E_3$ and $\{v_1, v_2\} \in E_3$.

Now, due to immoralities, G_1 forbids v_2 from being the last in the ordering, G_2 forbids v_3 from being the last in the ordering and G_3 forbids v_1 from being the last in the ordering. Therefore it is not possible to arrive to each of these solutions with a mutual vertex ordering. □

3.3 Dynamic choice construction

In this section we will propose a dynamic programming approach to branch selection and parent set pruning during search, based on the following definition of valid parent sets.

Definition 18. *Let $G = (V, E, \pi)$ be an ordered decomposable DAG. Given $v_k \notin V$ and $P \subseteq V$, let $G' = (V', E', \pi')$ be G with the parent set choice (v_k, P). The parent set choice (v_k, P) is* valid *for G if the following hold.*

1. *For all $v_i, v_j \in P$ we have either $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$. (Forbid immoralities)*
2. *For all $v_i \in V$, the pair (v_i, v_k) does not violate the preferred vertex order in G'. (Definition 16)*
3. *The depths of G' are ordered. (Definition 15)*

Sometimes we are given an instance with a bounded treewidth, which means that we are restricted to using parent sets that have less elements than a given threshold. In this case the Definition 18 (valid parent set choice) must include the size restriction.

Given a partial solution $G = (V, E, \pi)$, a vertex $v \notin V$, and a subset $P \subseteq V$, function GetSupersets in Algorithm 2 represents a dynamic programming method for determining valid parent set choices (v, P') for G where $P' \supseteq P$. An advantage of this formulation is that invalidating conditions for a parent set, such as immoralities or violations of the preferred vertex order, automatically hold for all the supersets of the parent set; this is applied on line 8 to avoid unnecessary branching.

On line 12 we require that a parent set P is added to the list only if none of its valid supersets $P' \in \mathcal{C}$ have a higher score. This pruning technique is based on the observation that P' provides all the same moralizing edges as P, and therefore it is sufficient to only consider the parent set choice (v, P') in the search when $s(v, P) \leq s(v, P')$.

Algorithm 2 Constructing parent set choices via dynamic programming.

```
1: function ParentSetChoices(U, G = (V, E, π))
2:     return ⋃_{v∈U} ⋃_{M∈M(G,v)} GetSupersets(v, G, M)
3: end function
4:
5: function GetSupersets(v, G = (V, E, π), P)
6:     Let C ← ∅
7:     for v' ∈ V \ P \ {v} do
8:         if (v, P') is valid for G with some P' ⊇ P ∪ {v'} then
9:             C ← C ∪ GetSupersets(v, G, P ∪ {v'})
10:        end if
11:    end for
12:    if (v, P) is valid for G and s(v, P) > s(v, P') for all P' ∈ C then
13:        C ← C ∪ {(v, P)}
14:    end if
15:    return C
16: end function
```

Also the symmetry-breaking constraints behave in such way that it is safe to choose a higher-scoring valid parent set over its subsets. For example, the condition 3 in Definition 16 is only less likely to hold when the set $pa_G(v_i)$ contains additional elements. Furthermore, the pruning on line 12 does not force any particular order for the graph construction, and as such it does not affect the constraint of depths being ordered.

Given the set of remaining vertices U, function ParentSetChoices in Algorithm 2 constructs all the available parent set choices for the current partial solution $G = (V, E, \pi)$. The collection $\mathcal{M}(G, v_i)$ contains the *subset-minimal parent sets* for vertex $v_i \in U$ that satisfy the third condition of Definition 18. If $V = \emptyset$, then $\mathcal{M}(G, v_i) = \{\emptyset\}$. Otherwise, let k be the maximum depth of the vertices in G. Now $\mathcal{M}(G, v_i)$ contains the subset-minimal parent sets that would insert v_i on depth $k + 1$. In addition, if $i > j$ for all $v_j \in V$ where $d(G, v_j) = k$, then $\mathcal{M}(G, v_i)$ also contains the subset-minimal parent sets that would insert v_i on depth k. Note that the cardinality of any parent set in $\mathcal{M}(G, v_i)$ is at most one.

3.4 Bounds for CMSL

In this section we introduce methods for computing strong upper and lower bounds for chordal Markov network structure learning.

Algorithm 3 A dynamic-programming algorithm for constructing a segment tree for the given scoring function. [30]

```
1: function OptimalParentSet(v, V)
2:     if V = ∅ then return ∅
3:     P ← V
4:     for v' ∈ V do
5:         P' ← OptimalParentSet(v, V \ {v'})
6:         if s(v, P) < s(v, P') then P ← P'
7:     end for
8:     return P
9: end function
```

3.4.1 BNSL-based upper bounds

When we use a same scoring function for both BNSL and CMSL, the score of the obtained optimal Bayesian network structure is always an upper bound value for the score of the optimal decomposable DAG. For this reason we can use BNSL to obtain upper bounds for CMSL.

We will start by explaining how to construct a *segment tree* for a given scoring function, since this will be needed when computing bounds for CMSL. Segment tree is a data structure that allows one to store intervals for efficient access [9].

Given a scoring function s, a vertex v and a set of vertices $v \notin V$, Algorithm 3 determines the highest-scoring parent set $P \subseteq V$ for v [30]. Formally,

$$\textsc{OptimalParentSet}(v, V) = \arg\max_{P \subseteq V} s(v, P).$$

When V is empty, the optimal parent set is trivially the empty set (line 2). Otherwise, we will find the optimal parent set by recursively going through all the subsets of V as follows. On line 3 we initialize an auxiliary variable P, which denotes the incumbent optimal parent set of v within V. Inside the loop of line 4, we check all the ways of removing a single vertex from V, and recursively call OptimalParentSet to obtain the optimal parent sets for these subsets. If any of these parent sets P' provide a higher score than the incumbent parent set P, we update P to equal P'. When the loop is terminates, P will be the highest-scoring parent set for v within V.

It is clear that we need to use dynamic programming to make Algorithm 3 efficient. But, on the other hand, we will be calling Algorithm 3 repeatedly while computing upper and lower bounds. For this reason we suggest that the output values of OptimalParentSet would be kept cached, not only

Algorithm 4 A dynamic programming algorithm to calculate the optimal Bayesian network structure score [30].

```
1: function OptimalBayes(V, A)
2:     if A = V then return 0
3:     Let b ← −∞
4:     for v ∈ V \ A do
5:         Let P = OptimalParentset(v, A)
6:         b ← max(b, s(v, P) + OptimalBayes(V, A ∪ {v})
7:     end for
8:     return b
9: end function
```

during its own execution, but throughout the entire branch and bound search. That is, given a set of vertices V of the CMSL instance, we can call OptimalParentSet$(v, V \setminus \{v\})$ for each vertex $v \in V$ in the beginning of the branch and bound, and then have all the optimal parent set information stored in the DP tables which work as segment trees. Thus, when n is the number of variables of the CMSL instance, we will use $\Theta(n \cdot 2^n)$ time to construct the segment trees, and then the following calls to Algorithm 3 will only take a constant time, since they will simply fetch the already-computed values from memory.

To compute an optimal Bayesian network structure, we will use a standard dynamic algorithm by Silander and Myllymäki [30]. Our motivation for using dynamic programming is that it automatically stores the optimal BN structure scores for all the subsets of the vertices. That is, we use same strategy as in the case of OptimalParentset. Thus by spending $\Theta(n \cdot 2^n)$ time in the beginning of the branch and bound, we will obtain all the BNSL-based upper bounds for CMSL in $\mathcal{O}(1)$ time by reading the DP table during search [30].

The DP-based BNSL works by recursively modifying an implicit BN structure B and scoring it. As an input we are given a set of variables V of the BNSL/CMSL instance, and also a subset $A \subseteq V$ that we call *assigned vertices*. The assigned vertices represent the current contents of B. Then our task is to find the highest-scoring way of adding the unassigned vertices $V \setminus A$ to B. Note that since we are learning BN structures here, immoralities can be ignored. For this reason any unassigned vertex $v \in V \setminus A$ could potentially use any subset $P \subseteq A$ as its parent set when added to B.

Given the sets V and A, Algorithm 4 returns the score of an optimal way of choosing parent sets for the unassigned vertices $V \setminus A$. When $A = V$, we have the base case, and return 0 as the optimal score. Otherwise we use a variable b

to store the incumbent highest score. In the loop on line 4 we go through each unassigned vertex $v \in V \setminus A$ as a means of checking how high BN structure scores we would get by now adding v to B. First, we use OptimalParentset to determine the highest-scoring parent set $P \subseteq A$ for v. Then, we call the algorithm recursively to evaluate $s(v, P) + \text{OptimalBayes}(V, A \cup \{v\})$. If this value is higher than b, we update b to equal this. In the end, when the loop is finished, b is the optimal score for the parent set choices of the unassigned vertices.

Now, calling $\text{OptimalBayes}(V, \emptyset)$ will let us store the scores of the optimal BN structures over the subsets of V. As with Algorithm 3, we will do this exhaustive call in the beginning of the branch and bound, and afterwards assuming calls to OptimalBayes taking unit time.

The $\Theta(n \cdot 2^n)$ time complexity of the upper bound computation might seem unefficient at first. However, for perspective, Junctor uses $\Omega(4^n)$ time to solve an entire CMSL instance [20], and our branch and bound approach for CMSL is not *theoretically* better than that. It is clear that $\Omega(4^n)$ grows a lot faster than $\Theta(n \cdot 2^n)$. For instance, at $n = 12$, we have $\frac{n \cdot 2^n}{4^n} = 0.0029296875$. In other words, as CMSL appears to be computationally a much harder problem than BSNL, we can afford to solve related BSNL instances exactly in the process of solving a CMSL instance. Empirically we report that, at $n = 20$, we are able to calculate the optimal BN structures in less than a minute using Algorithm 4.

3.4.2 Improved upper bounds

In Section 3.4.1 we explained how to calculate simple BNSL-based upper bounds for CMSL. The introduced method required us to compute the optimal Bayesian networks for all the subproblems, which could then be used as upper bounds for CMSL. In this section we explain how we can at times achieve even better upper bounds.

The upper bounds obtained via BNSL can be at times can quite loose when the networks contain a lot of immoralities. For this reason, in Algorithm 5, we introduce an additional method for computing the upper bounds, taking immoralities "relaxedly" into consideration. The algorithm takes four inputs: A fixed partial solution $G = (V, E, \pi)$, a list of vertices A that we have assigned during the upper bound computation, a list of remaining vertices U, and an integer $d \geq 0$ which dictates the maximum recursion depth. As a fallback option, on line 2 we return the optimal BN score for the remaining vertices if the maximum recursion depth is reached.

On line 3 of Algorithm 5 we construct the collection of sets $\mathcal{P}$ that are the maximal sets that any vertex can take as parent set during the upper bound computation. The sets in $\mathcal{P}$ take immoralities relaxedly into consideration: For any $v_i, v_j \in V$, we have $\{v_i, v_j\} \subseteq P$ for some $P \in \mathcal{P}$ if and only if $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$. That is, when choosing parent sets during the upper bound computation, we allow immoralities to appear, as long as they are not between vertices of the fixed partial solution.

In the loop on line 5, we iterate through each vertex $v \in U$ that is still remaining, and find its highest-scoring relaxedly-moral parent set according to $\mathcal{P}$. Note that given any $P' \in \mathcal{P}$, we can find the highest-scoring parent set $P \subseteq P'$ in $\mathcal{O}(1)$ time using the constructed segment tree from Section 3.4.1. Thus line 6 takes $\mathcal{O}(|V|)$ time to execute. Finally, on line 7 of the loop, we split the problem into subproblems to see which parent set choice (v, P) provides the highest local upper bound u to be returned.

We can often reduce the collection $\mathcal{P}$ by removing each element that is a proper subset of some other element in $\mathcal{P}$. This is because it is sufficient to consider $\hat{P} \in \mathcal{P}$ instead of $P' \in \mathcal{P}$ on line 6 if $P' \subset \hat{P}$. Note that, due to the morality, each term $\{v\} \cup pa_G(v)$ on line 3 is actually a clique in the fixed partial solution. For this reason, in our implementation, we find the collection of maximal cliques $\bigcup_{v \in V} \{\{v\} \cup pa_G(v)\} \subseteq \mathcal{C}$ of the partial solution in the beginning of the upper bound computation, and then have $\mathcal{P} = \bigcup_{C \in \mathcal{C}} \{C \cup A\}$. This speeds the algorithm up by a rather small margin, and for this reason a simplified version of $\mathcal{P}$ was chosen for the pseudocode.

Example 8. *Consider Figure 6. Here we see a set of relaxedly moral vertices $A = \{v_{10}, v_{11}\}$, and a fixed partial solution with the vertices $V = \{v_1, v_3, v_4, v_6, v_7, v_8\}$. The vertex v_{11} has both v_{10} and v_7 as its parents, even though there is no moralizing edge between v_{10} and v_7. This is allowed because the upper bound computation only forbids immoralities where both parents belong in V.*

If we were to assign a new relaxedly moral vertex into Figure 6, its parent set would be a subset of one of the sets in $\mathcal{P}$:

$$\mathcal{P} = \{\{v_1, v_{10}, v_{11}\}, \{v_1, v_3, v_{10}, v_{11}\}, \{v_1, v_3, v_4, v_{10}, v_{11}\}, \\ \{v_6, v_{10}, v_{11}\}, \{v_6, v_7, v_{10}, v_{11}\}, \{v_6, v_7, v_8\, v_{10}, v_{11}\}\},$$

which can be reduced to

$$\mathcal{P} = \{\{v_1, v_3, v_4, v_{10}, v_{11}\}, \{v_6, v_7, v_8, v_{10}, v_{11}\}\}.$$

For comparison, Algorithm 4 (dynamic programming for BNSL), could choose any subset of $V \cup A = \{v_1, v_3, v_4, v_6, v_7, v_8, v_{10}, v_{11}\}$ in the same situation.

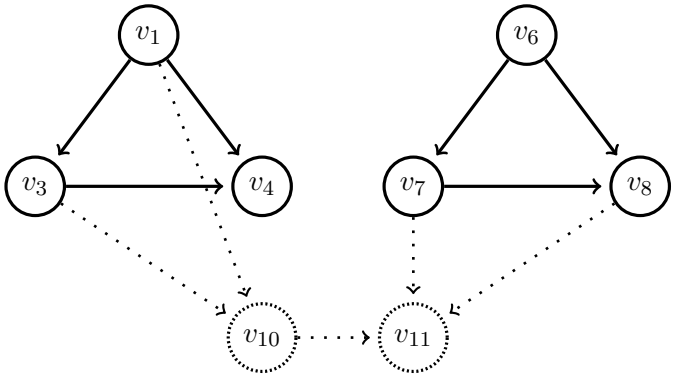

Figure 6: An example scenario in Algorithm 5. The vertices and edges with solid lines represent a fixed partial solution, and the vertices and edges with dotted lines represent a set of relaxedly moral vertices, i.e. the set A.

Algorithm 5 requires $\mathcal{O}((n-m) \cdot m \cdot 2^{n-m})$ time, where $m = |V|$ is the number of vertices in the fixed partial solution and n is the total number of vertices in the problem instance. This analysis assumes that the optimal BN structures and the segment trees have been precomputed. (In the empirical evaluation, the total runtimes of our branch and bound approach include these computations.)

We use the upper bounds within the branch and bound as follows. Let G be the current partial solution, let U be the set of remaining vertices, and let B be the optimal Bayesian network structure for the vertices in U. We can close the current branch if $s(G^*) \geq s(G) + s(B)$. Otherwise, we can close the branch if $s(G^*) \geq s(G) + \textsc{UpperBound}(G, \emptyset, U, d)$ for some $d > 0$. Our implementation uses $d = 10$.

Unlike in the case of Algorithm 4 (computing the optimal BN structures), here we would not benefit from keeping the DP table stored for later use. This is because here the returned values do not depend only on the vertices of the partial solution G, but also on the *edges* between those vertices. This means that changing even a single edge in the partial solution could potentially result in different outcome in Algorithm 5. Therefore we will call the algorithm repeatedly in the branch and bound.

Algorithm 5 requires a lot of time considering how often it is used. For this reason we will now propose couple of ways to make it more efficient.

Identifying when the BNSL score is unlike to be improved. It is clear that an optimal BNSL score is an upper bound value to the corresponding score given by UpperBound. However, if the current partial solution (graph) is complete, i.e. each pair of vertices in the graph is connected, then the

Algorithm 5 Computing upper bounds for a partial solution via dynamic programming.

```
1: function UpperBound(G = (V, E, π), A, U, d)
2:     if d = 0 then return OptimalBayes(U)
3:     Let P = ⋃_{v∈V} {{v} ∪ pa_G(v) ∪ A}
4:     Let u ← −∞
5:     for v ∈ U do
6:         Let P = arg max_{P⊆P'∈P} s(v, P)
7:         u ← max(u, s(v, P) + UpperBound(G, A ∪ {v}, U \ {v}, d − 1))
8:     end for
9:     return u
10: end function
```

BNSL score and the score by UpperBound will be equal. This is because no immoralities could now occur between the vertices of the partial solution.

Following the same intuition, the larger the parentsets are in the partial solution, the less restrictive UpperBound is, and thus the closer its results might be to that of the BNSL score. For this reason it might be reasonable to only use UpperBound when the density of the partial solution is not too high. It is also possible to retrieve the corresponding Bayesian network structure using the DP table and to check whether it contains immoralities. If it does not, then UpperBound could not possibly find a graph with lower score.

Terminating Algorithm 5 early. In the branch and bound, UpperBound is only used to check whether its result is greater or equal to $s(G^*) - s(G)$. Therefore it might be useful to pass this value to UpperBound, so that the upper bound calculation can be terminated as soon as any value greater or equal to the threshold is found.

3.4.3 Lower bounds

In this section we introduce a method for calculating lower bounds for CMSL. A good initial lower bound is an essential part of our algorithm since it may help us prune large parts of the search space from early on.

Our approach is based on first constructing an (infeasible) upper bound solution and then turning it into a feasible lower bound solution. We can use the optimal Bayesian network structures (as described in Section 3.4.1) as the upper bounds.

Algorithm 6 A way to construct the optimal BN structures when the scores of those structures have already been computed [30].

```
1: function ConstructBNStructure(V)
2:     Let V' ← ∅
3:     Let E' ← ∅
4:     while V' ≠ V do
5:         v ← arg max_{v∈V\V'} OptimalBayes(V, V' ∪ {v})
6:         P ← OptimalParentset(v, V')
7:         V' ← V' ∪ {v}
8:         E' ← E' ∪ {(p, v) : p ∈ P}
9:     end while
10:    return G = (V', E')
11: end function
```

With the DP tables, it takes only a low polynomial time to retrieve the optimal BN structure for a given set V of vertices [30]. A straight-forward method for achieving this is shown in Algorithm 6. Here, the variables V' and E' store the vertices and edges of the graph to-be-constructed, respectively. On line 5 we check which vertex $v \in V \setminus V'$ yielded the highest-scoring BN structure when the set of assigned vertices was $V' \cup \{v\}$. On line 6 we check which parent set $P \subseteq V'$ yielded the highest score for variable v. Then, we use v and P to update V' and E'. We repeat this process as long as V' does not contain all the vertices in V; thus the loop takes exactly $|V|$ iterations to complete.

Now, we can obtain an initial lower bound solution G^* for the branch and bound as follows.

1. Construct the optimal BN structure over all the vertices of the given CMSL instance.
2. Heuristically try different ways to make the structure decomposable by either adding or removing edges.
3. Let G^* be the highest-scoring decomposable DAG from Step 2.

There are various ways to implement Step 2 (graph moralization). In general finding a way to turn non-chordal graph into chordal graph by adding a minimum number edges is a NP-hard problem [15, 14]. Since chordal graphs are equivalent to decomposable DAGs, this means that in general finding a score-optimal way to turn non-decomposable DAG into decomposable DAG is NP-hard as well. For this reason our implementation settles for finding an approximate solution instead. This does not affect the correctness of the overall search algorithm.

When we are using a bounded treewidth, it is impossible to moralize certain graphs by simply adding new edges to it. This happens when the moralization requires us to add new parents for a vertex that had already reached the maximum parent set size. For this reason our implementation uses two moralization schemes; (1) only adding edges and (2) only removing edges, and chooses the highest-scoring graph out of the obtained solutions. The latter method (removing edges) is guaranteed to always give a feasible solution, since a graph with no edges is trivially decomposable.

Given a moralization scheme, we can use a greedy algorithm to moralize a graph. In this case we go over each of the immoralities in the graph, and then either (1) look for addable edges that would remove the immorality, or (2) look for removable edges that had formed the immorality. In both cases we select the edge that locally worsens the score of the resulting graph as little as possible. Adding/removing edges can cause more immoralities to appear, and as such the result of the locally optimal choices can be far from the globally optimal one.

Our method for moralization works as follows. First we consider all the non-cyclic outcomes after adding/removing at most a given number of edges. Then we proceed to moralize those outcomes greedily, all while keeping track of the highest-scoring moralized graph. It should be noted that it is possible to arrive to same graph multiple times during the procedure. We can use caching to store the visited graphs, and this way avoid doing unnecessary work.

The above procedure for obtaining the initial lower bound can also be extended to the rest of the search. That is, on any node of the search tree in the branch and bound, we can merge the current partial solution with the Bayesian network structure that corresponds to the node, and then moralize the result to get a new feasible solution. Empirically, it is not uncommon for the optimal solution to be found this way, but the constant moralization takes such time that it does not generally pay off. However, this technique might become useful if a good heuristic was developed to judge whether moralizing a given graph could improve the current lower bound.

3.5 Example run of the algorithm

This section provides an example of how our branch and bound operates on a selected problem instance.

Let the vertices of the problem instance be $V = \{v_1, v_2, v_3, v_4, v_5\}$. Even for a very small number of variables like this, the search process is very complicated

considering the dynamic branch selection (Section 3.3), bound computations (Section 3.4.1, 3.4.2 and 3.4.3) and constraint evaluation (Section 3.2) in every search tree node. For this reason we present a very high-level example of the overall search process where all unnecessary details are omitted for readability. This is also the reason why we will not list all the $5 \cdot 2^4 = 80$ parent set scores here.

Before we start the actual branch and bound search (Section 3.1), we use dynamic programming to compute an upper bound solution B, which is an optimal Bayesian network structure over V (Section 3.4.1). Assume that we obtain the graph (a) in Figure 7.

The next step is to compute an initial lower bound solution G^*, which is obtained by greedily moralizing B (Section 3.4.3). Assume that the result is graph (b) in Figure 7. In this case adding the edge $v_1 \to v_2$ yielded the highest-scoring outcome in the greedy moralization.

We are now ready to begin the branch and bound search (Section 3.1). We will keep track of a partial solution $G = (V', E, \pi)$ to which we gradually add new parent set choices (Definition 13). However, as we explained in Section 3.2, the constraint of preferred vertex order forces us to fix the lexicographically smallest vertex as a source node. Assume that we start with $V' = \{v_1\}$, $E = \emptyset$, and $\pi(1) = 1$.

We are at the root node of the search tree. Here the set of possible parent set choices is

$$\{(v_i, \emptyset) : v_i \in V \setminus \{v_1\}\} \ \cup \ \{(v_i, \{v_1\}) : v_i \in V \setminus \{v_1\}\}.$$

That is, we could add one of the remaining vertices $\{v_2, v_3, v_4, v_5\}$ to G, and each vertex could have either $\emptyset$ or $\{v_1\}$ as its parent set, resulting in total of

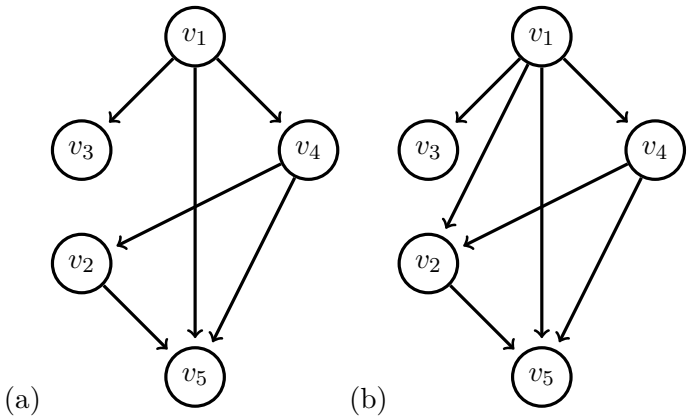

Figure 7: Examples of initial upper and lower bound solutions, respectively.

eight possibilities. Note that in this situation it is not possible to violate the symmetry-breaking rules of Section 3.2.

When we use dynamic programming to determine the necessary parent set choices (Section 3.3), we apply the on-the-fly score pruning along the way. Assume that we have $s(v_i, \emptyset) \leq s(v_i, \{v_1\})$ for each $v_i \in \{v_2, v_3, v_4, v_5\}$. Now, it is sufficient to try only four parent sets in this search tree node, namely $(v_5, \{v_1\})$, $(v_3, \{v_1\})$, $(v_2, \{v_1\})$ and $(v_4, \{v_1\})$. We will try these choices in ascending order based on their scores.

Assume that we construct a new partial solution that is G with the parent set choice $(v_5, \{v_1\})$ (Definition 13), and move to the respective search tree node (Section 3.1). However, as we calculate upper bound for this branch (Section 3.4.2), we notice that it is less than our current lower bound $s(G^*)$. Thus we conclude that the current branch does not lead to an improved solution, and so we backtrack in the search (Section 3.1).

Assume that we construct a new partial solution that is G with the parent set choice $(v_3, \{v_1\})$ (Definition 13) and move to the respective search tree node (Section 3.1). This time we notice that the computed upper bound (Section 3.4.2) is higher than the current lower bound $s(G^*)$. Therefore we will continue exploring in this branch.

Continuing with the search, assume that we have added the parent set choices $(v_4, \{v_1\})$, $(v_5, \{v_1, v_4\})$ and $(v_2, \{v_4, v_5\})$ to G, resulting in a solution G'. The gradual construction of G' can be seen in Figure 8. Since G' contains all the vertices in V, we are now in a leaf node in the search tree. We notice that $s(G^*) < s(G')$, i.e., G' has higher score than our current lower bound solution G^*. Thus we update $G^* \leftarrow G'$ and backtrack in the search (Section 3.1).

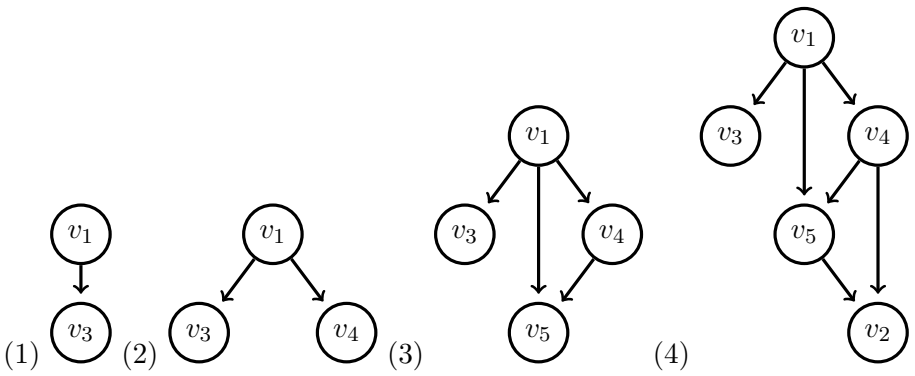

Figure 8: An example of how a solution is gradually constructed during the search.

Assume that the higher lower bound $s(G^*)$ now causes all the remaining branches to be closed since none of the computed upper bounds are high enough (Section 3.4.2). In this case G^* (i.e. G') is an optimal solution to the problem instance, and its skeleton is a score-optimal chordal Markov network over V with respect to the given scoring function.

4 Empirical evaluation

In this section we will empirically evaluate the practical performance of the branch-and-bound algorithm proposed in this work.

4.1 Experiment setup

To recap from Section 2.5: GOBNILP [32, 2] is a state-of-the-art Bayesian network structure learning system that implements an integer programming branch-and-cut approach that allows CMSL by ruling out DAGs that are not decomposable. Junctor [20, 21] is a state-of-the-art dynamic programming approach to CMSL. We implemented our branch and bound algorithm in C++, and will refer to this prototype as *BBMarkov*. The source code of the implementation can be found at http://cs.helsinki.fi/group/coreo/bbmarkov.

We used a total of 54 real-world datasets used as standard benchmarks for exact approaches [41, 38]. For investigating scalability of the algorithms in terms of the number of variables n, we obtained from each dataset several benchmark instances by restricting to the first n variables for increasing values of n. This strategy also allowed us to include tests with datasets whose variable counts would otherwise be too large for CMSL.

We follow a standard practice of benchmarking exact structure learning algorithms and do not include the time spent for constructing the scoring functions. That is, we give the pre-computed scoring function (i.e. a list of clique or parent set scores) as an input to the algorithms, and then measure how much time they need to terminate with an optimal solution. We used the BDeu scoring function with equivalent sample size 1.

We will consider instances with both unbounded and bounded treewidths. Bounding the treewidth means restricting the maximum parent set size for decomposable DAGs in BBMarkov and GOBNILP, and restricting the maximum clique size for clique trees in Junctor.

All the experiments were run using Debian GNU/Linux on 2.83-GHz Intel Xeon E5440 computer with 32-GB RAM. We used GOBNILP version 1.6.2 with IBM ILOG CPLEX Optimization Studio (version 12.7.1.0) as the linear programming solver.

Table 2: Comparison between BBMarkov and Junctor when solving full datasets, and the running times for the 17-variable *water* and *carpo* datasets under different sample sizes. The numbers in parentheses indicate the time BBMarkov needed to find its final (optimal) solution.

		Running times (s)		
Dataset	n	**BBMarkov**		**Junctor**
Wine	13	<1	(<1)	6
Adult	14	58	(35)	29
Letter	16	>3600	(>3600)	592
Voting	17	281	(207)	3050
Zoo	17	>3600	(>3600)	2690
Tumor	18	610	(268)	12019
water100	17	100	(49)	2580
water1000	17	2731	(279)	2592
water10000	17	>3600	(>3600)	2928
carpo500	17	1625	(979)	2581
carpo1000	17	3080	(3067)	2586
carpo10000	17	>3600	(>3600)	2623

4.2 Running time comparison

In this section we compare the running times of BBMarkov to those of Junctor and GOBNILP.

Figure 9 compares the running times of BBMarkov and GOBNILP when the treewidths of the instances is not restricted. A 1-hour time limit was used for each instance, which is represented with the grey dashed line. The different variable counts (n-values from 11 to 15) of the instances are represented by using different symbols for each n. The vast majority of the points of the plot are located on the left side of the diagonal, which indicates that BBMarkov is in general faster than GOBNILP for CMSL. Tests for $n > 15$ are omitted in the plot since GOBNILP was unable to solve any of such instances within the time and memory limit.

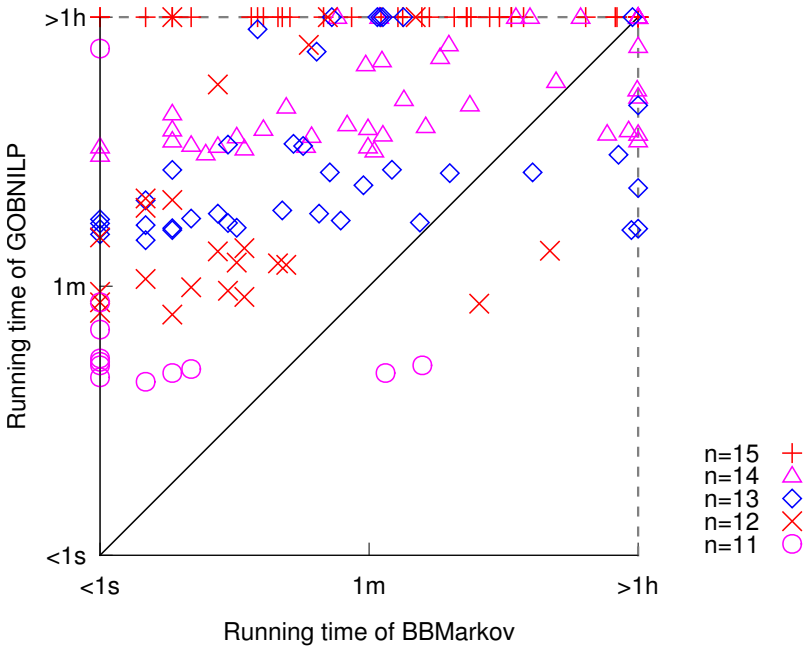

Figure 9: Running time comparison between BBMarkov and GOBNILP. The dashed line denotes the 1-hour timeout or memout.

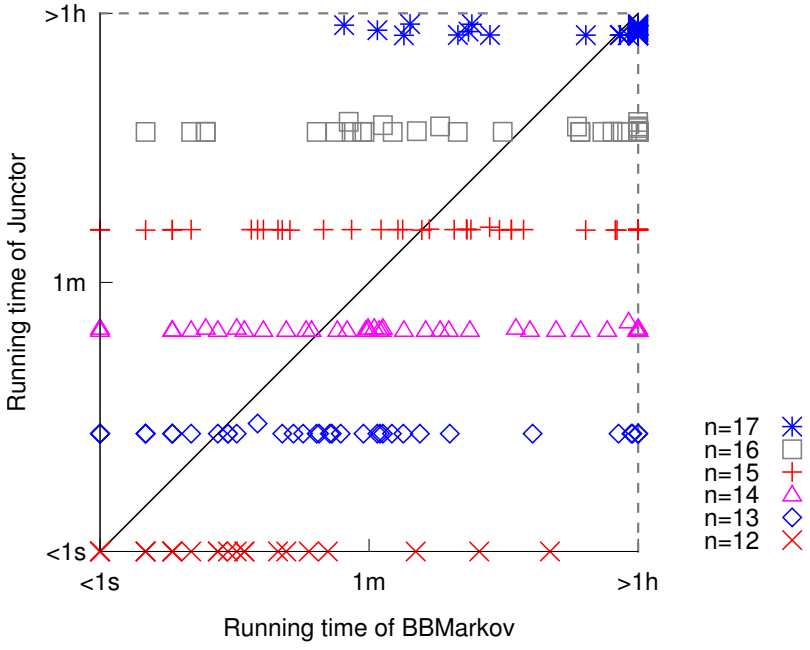

Figure 10: Running time comparison between BBMarkov and Junctor. The dashed line denotes the 1-hour timeout or memout.

Table 3: Examples of best-case performance of BBMarkov with large number of variables. The numbers in parentheses indicate the time BBMarkov needed to find its final (optimal) solution.

		Running times (s)		
Dataset	n	**BBMarkov**		**Junctor**
alarm	17	268	(62)	2724
	18	1462	(315)	12477
	19	10274	(2028)	52130
	20	49610	(50)	**memout**
Heart	17	41	(22)	3007
	18	162	(85)	11179
	19	1186	(698)	50296
	20	15501	(13845)	**memout**
insurance100	17	113	(28)	3056
	18	494	(113)	11334
	19	17125	(15150)	50014
	20	>86400	(>86400)	**memout**
hailfinder500	17	225	(108)	2588
	18	2543	(1348)	12422
	19	13749	(6418)	53108
	20	33503	(25393)	**memout**
mildew1000	17	379	(287)	2590
	18	1751	(1278)	11294
	19	33160	(27421)	52572
	20	>86400	(>86400)	**memout**
water100	17	100	(49)	2580
	18	590	(244)	12244
	19	6581	(6187)	52575
	20	61152	(54806)	**memout**

Figure 10 compares BBMarkov to Junctor when treewidth is not restricted. The scatter plot follows the exact same format as Figure 9, except that here the n-values (variable counts) vary from 12 to 17. The tests with $n > 17$ were omitted from the plot since Junctor was unable to solve any of them under the 1-hour time limit. For this reason there are no timeouts for Junctor in this plot.

We can see from Figure 10 that the running times of Junctor are consistent with a same n-value. For any $n = 17$ instance, we know that we are able to solve it within 1 hour with Junctor, whereas there is no guarantee whether the same holds true for BBMarkov. On the other hand, for any $n = 18$ instance, BBMarkov might be able to solve it within 1 hour, whereas it is guaranteed that Junctor cannot (with the hardware used in our tests). It should be noted that this conclusion is not only based on empirical data but also on the theoretical $\Omega(4^n)$ lower bound of Junctor.

Table 2 shows the running times of BBMarkov and Junctor when solving datasets whose variable count is not reduced from the original. The numbers in parentheses showcase the time that BBMarkov needed to find an optimal solution, whereas the numbers without surrounding parentheses denote overall running times. The table also showcases how the sample size of an instance may affect BBMarkov; the number of samples is the number displayed in the instance name.

As BBMarkov is very sensitive about different factors, e.g. how strong upperbounds can be calculated for a particular scoring function, it may perform especially well with some instances. Examples of this are shown in Table 3. A 24-hour time limit was used for these tests.

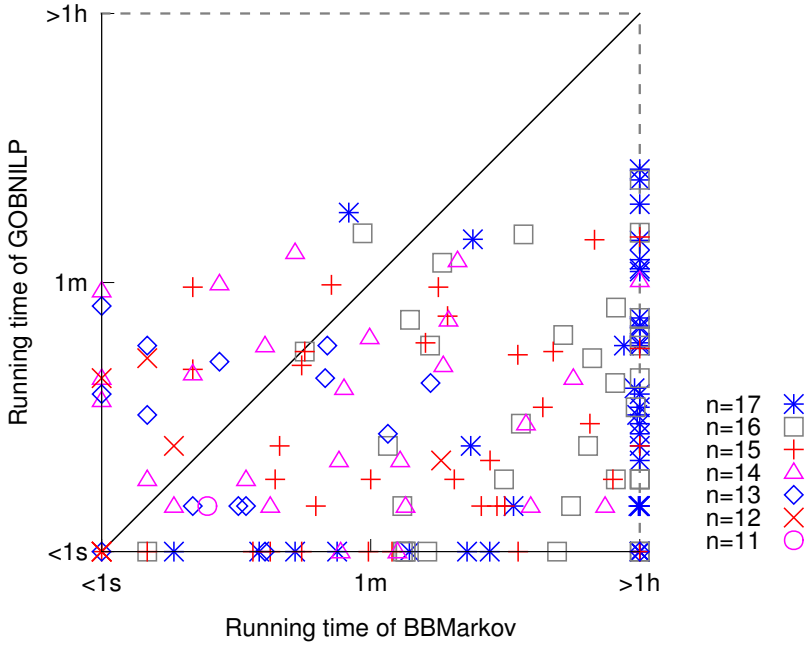

Figure 11: Comparison with GOBNILP when the treewidth is bounded to 2. The dashed line denotes the 1-hour timeout or memout.

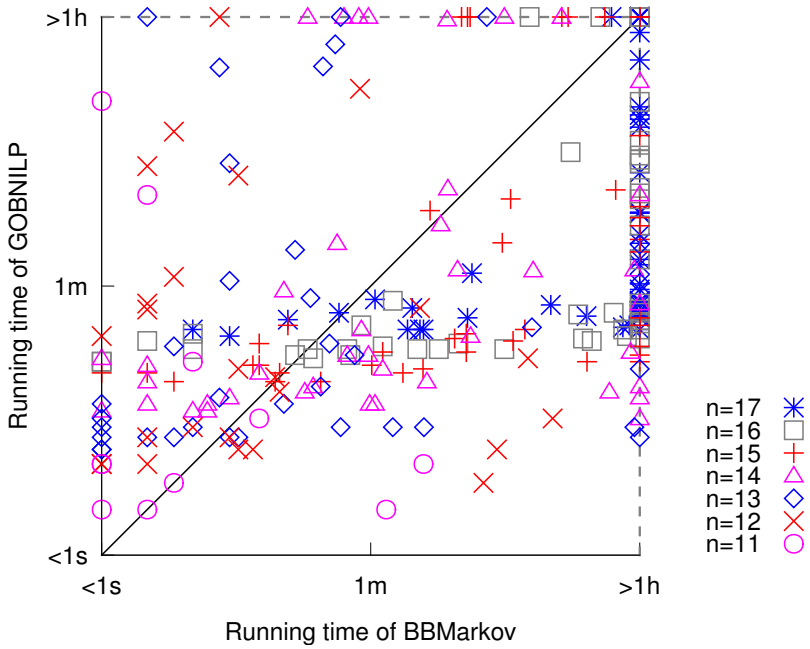

Figure 12: Comparison with GOBNILP when the treewidth is bounded to 4. The dashed line denotes the 1-hour timeout or memout.

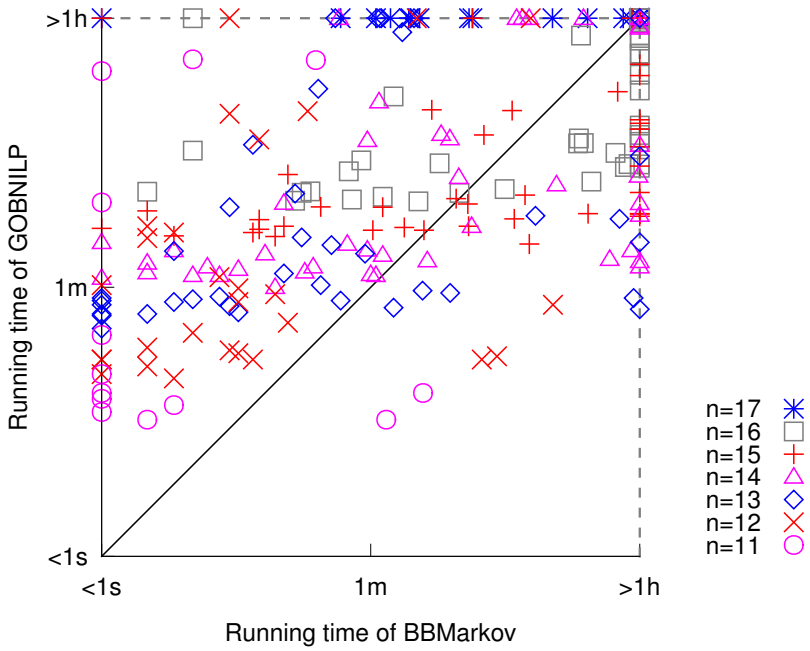

Figure 13: Comparison with GOBNILP when the treewidth is bounded to 6. The dashed line denotes the 1-hour timeout or memout.

Finally, we compare the running times of the algorithms when the treewidth of the instances is bounded. Figures 11, 12 and 13 compare BBMarkov to GOBNILP under treewidth bounds of 2, 4 and 6, respectively. Here we can see that GOBNILP benefits a lot from the treewidth-bounding, displaying clearly better performance compared to the unbounded case. We note that GOBNILP also becomes faster than Junctor when the treewidth bound is low enough on a sufficiently large number of variables [20]. As Figure 13 shows, BBMarkov is again faster than GOBNILP when treewidth bound is 6, but the difference is not as big as in the unbounded case. Judging by our empirical evaluation, sometimes our implementation solves an instance almost at same speed regardless of whether the treewidth is bounded or not. This is due to the greedy nature of how our branch and bound iterates branches in the search tree.

4.3 Memory usage and the impact of bounding

In this section we evaluate other aspects of BBMarkov beyond the running time comparison of previous section.

Figure 14 compares the memory usage of BBMarkov and Junctor with

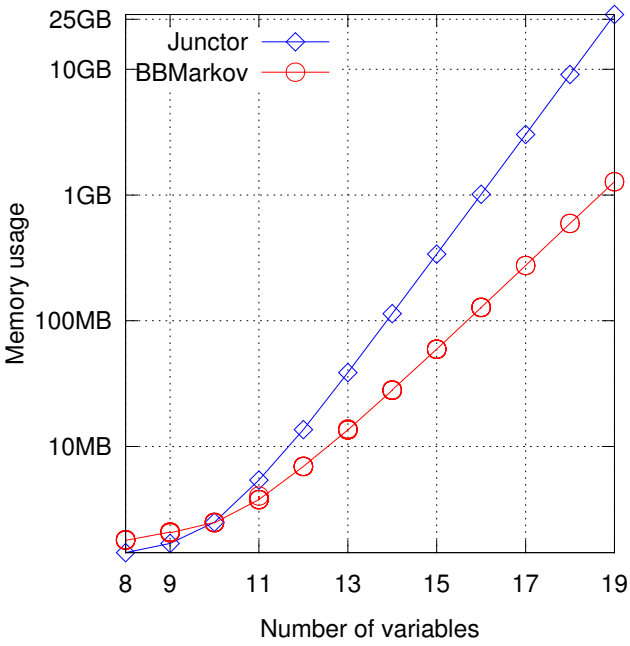

Figure 14: Comparison between the memory usage of BBMarkov and Junctor when the treewidth is not bounded.

unbounded treewidth. This evaluation was performed by solving a subset of the instances for each n and averaging the amount of used memory. Junctor's $\Omega(3^n)$ lower bound on memory usage makes it consistently run out of memory for $n \geq 20$. At $n = 19$, BBMarkov uses on average approx. 1 GB of memory whereas Junctor uses more than 25 GB. This is shown in Figure 14. The memory usage of GOBNILP is not examined here, but we report of numerous cases where the program ran out memory at $n = 16$. We note that GOBNILP uses considerably less memory with bounded treewidth, which was the reason why the evaluation in Figures 11, 12 and 13 was possible for $n \geq 16$.

Figure 15 compares the performance of BBMarkov to itself when using different strategies for computing upper bounds. Using optimal BN structure scores as upper bounds refers to the method described in Section 3.4.1. Similarly, using tight upper bounds refers to applying the method of relaxing immoralities as in Section 3.4.2. However, we do compute the optimal BN structures in both cases. This is because the BN scores are used to optimize the latter method (computing the tight upper bounds), and because we need the BN structures in order to calculate the initial lower bound like explained in Section 3.4.3.

As shown in Figure 15, using the tight bounds results in a better performance when the number of variables of the instance, n, is less than 15. When n increases, we can see that using only the BN-based bounds becomes a more efficient strategy. This is understandable, because even though the tight bounds might theoretically provide better upper bounds, they also become computationally rather expensive to calculate when the number of variables becomes large.

Compared to Junctor, one of the benefits of BBMarkov is its ability to print out lower bound solutions right after they are found during the search. When the execution of the algorithm finished, the latest lower bound is proven to be the optimal solution. For this reason it is not uncommon that the optimal solution is displayed well before the execution is terminated. In Figure 16, we can see comparison between the time needed for BBMarkov to merely find an optimal solution versus both finding and proving it.

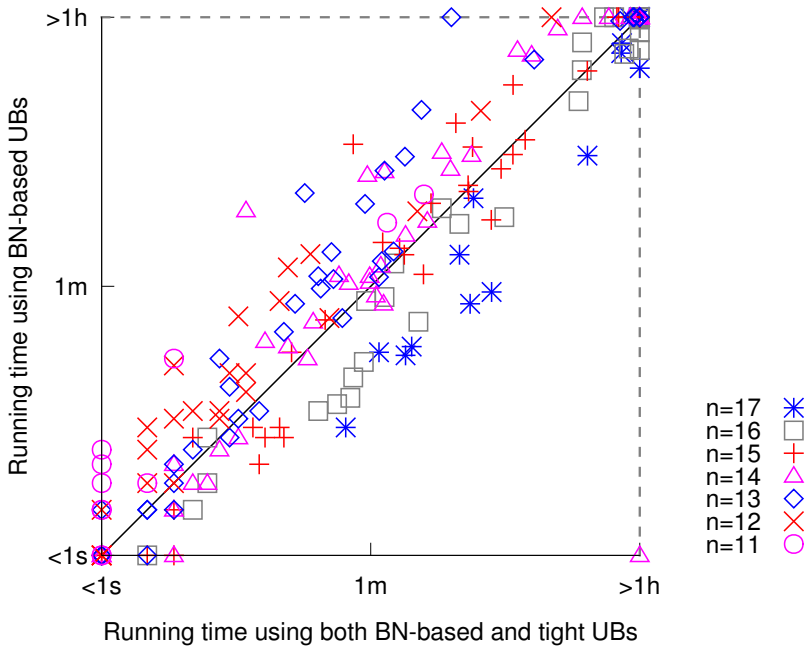

Figure 15: Comparison between two ways of computing upper bounds.

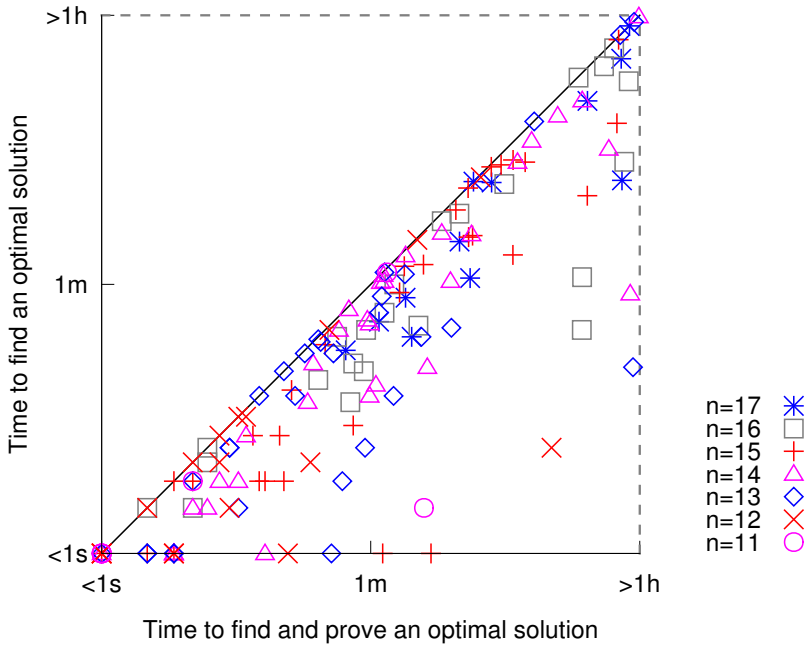

Figure 16: The difference in time for finding an optimal solution versus both finding and proving it.

5 Conclusions and further work

We introduced an exact branch and bound method for score-optimal chordal Markov network structure learning (CMSL). We showed how symmetry-breaking constraints can be used to detect equivalent solution candidates and thus considerably reduce the size of the search space. The symmetry-breaking technique was incorporated into a dynamic programming algorithm that, within each search tree node, determined the necessary choices for branching. We also showed how Bayesian network structure learning could be used to provide upper bounds for CMSL. In addition, we introduced a dynamic programming algorithm for calculating even tighter upper bounds by taking immoralities relaxedly into consideration.

Empirically, we showed that our approach (BBMarkov) was able to scale further in certain cases than the competing algorithms. When the treewidth was not bounded, BBMarkov was able to solve some 20-variable instances within 24 hours, whereas the same did not hold true for other algorithms. In addition, we noted that BBMarkov used considerably less memory than its counterparts; for example, GOBNILP ran out of memory at 16 variables and Junctor at 20 variables, whereas BBMarkov was very far from running out of memory at 20 variables. Overall, the techniques introduced in this thesis form a complementary approach to the earlier proposed state-of-the-art algorithms for learning score-optimal chordal Markov network structures.

We finish by proposing promising directions for future work.

Improved branching. One way of speeding up our approach would be introducing an improved branching strategy, i.e., a heuristic that aims at finding a good order for iterating the current search tree branches. This is important because it could help us find faster a solution that could potentially improve the incumbent lower bound solution. The improved lower bound solution, in turn, could help us close branches and terminate the search faster. The current implementation of BBMarkov uses a naive branching strategy of first trying the parent set choices that have the highest scores. An alternative branching strategy would be to sort the parent set choices based on the upper/lower bounds from the resulting partial solutions.

Improved bounding. In Section 3.2 we described constraints for pruning equivalent solutions based on symmetries. However, the upper bound computation of Algorithm 5 does not take symmetries into consideration in any way. Theoretically, if symmetry-breaking constraints were utilized in the algorithm, even if just relaxedly (like in the case of immoralities), stronger upper bounds would be obtained.

Parallelization. In general, relatively little research has been made on scaling up structure learning techniques by parallelization [35, 28]. Our algorithm does not currently utilize any parallelization techniques, either. When it is possible to explore multiple search tree branches simultaneously, getting stuck in a bad branch may not be as big concern. Therefore applying parallelization in the branch and bound could help us partially counter problems caused by less-than-optimal branching strategies.

Parallelizing the branch and bound would require that each thread would work with its own search tree node, which would lead them storing their own parent set choice lists. Since the parent set choice lists can be large, this would increase the algorithm's memory usage. However, since this would only add a constant factor to the memory requirement, based on our empirical evaluation the memory usage would still remain below that of Junctor and GOBNILP with a large number of variables, assuming a conservative number of threads.

Backwards search via lazy decomposability. The branch and bound constructs graph candidates by gradually adding new sink nodes to them. Currently the search process is performed in a "forward" manner, which means that the parent sets of the new sinks can contain any vertices *in* the graph as long as no immoralities are introduced along the way. It would also be possible to perform the search "backwards", which means that we would handle the decomposability lazily. That is, the parent sets of the new sinks could contain any vertices *not* in the graph as long as the immoralities introduced by earlier sink choices would eventually get resolved. For example, if the backwards search chooses $\{v_2, v_3\}$ as the parent set of v_1, then we would require that the future parent set choice of v_2 will contain v_3 or the future parent set choice of v_3 will contain v_2.

The advantage of a backward search would be the possibility of applying different kind of score pruning. In Section 3.2 we proposed on-the-fly score pruning for CMSL, which worked so that any parent set of a vertex could be pruned if it had a feasible *superset* with a greater or equal score. In a backwards search the rule would work in reverse, i.e., any parent set of a vertex could be pruned if it had a feasible *subset* with a greater or equal score. The latter idea is intuitively promising because it is closer to the score pruning performed in top BNSL approaches [38].

Incremental search. Given a set of random variables V and $v \in V$, it can be observed that an optimal chordal Markov network structure G_V for V can be very similar to an optimal structure $G_{V \setminus \{v\}}$ for $V \setminus \{v\}$. This observation is based on the optimal solutions obtained by solving instances with varying number of variables on same datasets during the empirical evaluation of Sec-

tion 4. The phenomenon is also intuitive, since the conditional dependencies between the variables in $V \setminus \{v\}$ are the same in both cases.

It would be possible to construct a greedy algorithm which determines an approximately optimal way of inserting v into $G_{V \setminus \{v\}}$ under a relaxing condition, e.g., without removing existing edges from the graph. Such a greedy algorithm could then be used to generate lower bounds for the optimal structure over V. We have already found a number of instances where this method results in considerably better lower bounds than the moralization of optimal BN structures, but further research is needed.

Based on the empirical evaluation and the super-polynomial time complexity of exact CMSL approaches, learning score-optimal chordal Markov network structure for variables $V \setminus \{v\}$ is multitudes faster than learning a structure over V. For example, in case of Junctor, the difference is always more than 4-fold in practice. For this reason we could afford to solve the problem instances incrementally for increasing numbers of variables, and use the solutions of preceding steps to generate (possibly very strong) initial lower bounds for the succeeding steps.

References

[1] H. J. Abel and A. Thomas. Accuracy and computational efficiency of a graphical modeling approach to linkage disequilibrium estimation. *Statistical Applications in Genetics and Molecular Biology*, 143(10.1), 2017.

[2] M. Bartlett and J. Cussens. Integer linear programming for the Bayesian network structure learning problem. *Artificial Intelligence*, 244:258–271, 2017.

[3] D. M. Chickering. A transformational characterization of equivalent Bayesian network structures. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence, Montreal, Quebec, Canada, August 18-20, 1995*, pages 87–98. Morgan Kaufmann, 1995.

[4] D. M. Chickering. Learning equivalence classes of Bayesian network structures. In *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence, Reed College, Portland, Oregon, USA, August 1-4, 1996*, pages 150–157. Morgan Kaufmann, 1996.

[5] J. Corander, T. Janhunen, J. Rintanen, H. J. Nyman, and J. Pensar. Learning chordal Markov networks by constraint satisfaction. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States.*, pages 1349–1357. NIPS, 2013.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[7] A. Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

[8] P. Dawid and S. L. Lauritzen. Hyper Markov laws in the statistical analysis of decomposable graphical models. *Annals of Statistics*, 21(3):1272–1317, 09 1993.

[9] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars. *Computational geometry: Algorithms and applications, 3rd Edition*. Springer, 2008.

[10] C. P. de Campos and Q. Ji. Properties of Bayesian Dirichlet scores to learn Bayesian network structures. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

[11] C. P. de Campos and Q. Ji. Efficient structure learning of Bayesian networks using constraints. *Journal of Machine Learning Research*, 12:663–689, 2011.

[12] P. Dellaportas and J. J. Forster. Markov chain Monte Carlo model determination for hierarchical and graphical log-linear models. *Biometrika*, 86(3):615–633, 1999.

[13] X. Fan and C. Yuan. An improved lower bound for Bayesian network structure learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 3526–3532. AAAI Press, 2015.

[14] F. V. Fomin and Y. Villanger. Subexponential parameterized algorithm for minimum fill-in. *SIAM Journal on Computing*, 42(6):2197–2216, 2013.

[15] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[16] P. Giudici and P. J. Green. Decomposable graphical Gaussian model determination. *Biometrika*, 86(4):785, 1999.

[17] P. J. Green and A. Thomas. Sampling decomposable graphs using a Markov chain on junction trees. *Biometrika*, 100(1):91, 2013.

[18] T. S. Jaakkola, D. Sontag, A. Globerson, and M. Meila. Learning Bayesian network structure using LP relaxations. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*, volume 9 of *JMLR Proceedings*, pages 358–365. JMLR.org, 2010.

[19] T. Janhunen, M. Gebser, J. Rintanen, H. J. Nyman, J. Pensar, and J. Corander. Learning discrete decomposable graphical models via constraint optimization. *Statistics and Computing*, 27(1):115–130, 2017.

[20] K. Kangas, M. Koivisto, and T. M. Niinimäki. Learning chordal Markov networks by dynamic programming. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 2357–2365. NIPS, 2014.

[21] K. Kangas, T. M. Niinimäki, and M. Koivisto. Averaging of decomposable graphs by dynamic programming and sampling. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence, UAI 2015, July 12-16, 2015, Amsterdam, The Netherlands*, pages 415–424. AUAI Press, 2015.

[22] M. Koivisto and K. Sood. Exact Bayesian structure discovery in Bayesian networks. *Journal of Machine Learning Research*, 5:549–573, 2004.

[23] D. Koller and N. Friedman. *Probabilistic Graphical Models - Principles and Techniques*. MIT Press, 2009.

[24] K. S. S. Kumar and F. R. Bach. Convex relaxations for learning bounded-treewidth decomposable graphs. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 525–533. JMLR.org, 2013.

[25] S. L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. In G. Shafer and J. Pearl, editors, *Readings in Uncertain Reasoning*, pages 415–448. Morgan Kaufmann Publishers Inc., 1990.

[26] G. Letac and H. Massam. Wishart distributions for decomposable graphs. *The Annals of Statistics*, 35(3):1278–1323, 2007.

[27] D. Madigan, J. York, and D. Allard. Bayesian graphical models for discrete data. *International Statistical Review/Revue Internationale de Statistique*, 63(2):215–232, 1995.

[28] A. L. Madsen, F. Jensen, A. Salmerón, H. Langseth, and T. D. Nielsen. A parallel algorithm for Bayesian network structure learning from large data sets. *Knowledge-Based Systems*, 117:46–55, 2017.

[29] B. M. Malone and C. Yuan. A depth-first branch and bound algorithm for learning optimal Bayesian networks. In *Graph Structures for Knowledge Representation and Reasoning - Third International Workshop, GKR 2013, Beijing, China, August 3, 2013. Revised Selected Papers*, volume 8323 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2013.

[30] T. Silander and P. Myllymäki. A simple approach for finding the globally optimal Bayesian network structure. In *UAI '06, Proceedings of the 22nd Conference in Uncertainty in Artificial Intelligence, Cambridge, MA, USA, July 13-16, 2006*. AUAI Press, 2006.

[31] N. Srebro. Maximum likelihood bounded tree-width Markov networks. *Artificial Intelligence*, 143(1):123–138, 2003.

[32] M. Studený and J. Cussens. Towards using the chordal graph polytope in learning decomposable models. *International Journal of Approximate Reasoning*, 88:259–281, 2017.

[33] J. Suzuki. Learning Bayesian belief networks based on the Minimum Description Length principle: An efficient algorithm using the B & B technique. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML '96), Bari, Italy, July 3-6, 1996*, pages 462–470. Morgan Kaufmann, 1996.

[34] J. Suzuki. An efficient Bayesian network structure learning strategy. *New Generation Computing*, 35(1):105–124, 2017.

[35] Y. Tamada, S. Imoto, and S. Miyano. Parallel algorithm for learning optimal Bayesian network structure. *Journal of Machine Learning Research*, 12:2437–2459, 2011.

[36] C. Tarantola. MCMC model determination for discrete graphical models. *Statistical Modelling*, 4(1):39–61, 2004.

[37] J. Tian. A branch-and-bound algorithm for MDL learning Bayesian networks. In *UAI '00: Proceedings of the 16th Conference in Uncertainty in Artificial Intelligence, Stanford University, Stanford, California, USA, June 30 - July 3, 2000*, pages 580–588. Morgan Kaufmann, 2000.

[38] P. van Beek and H. Hoffmann. Machine learning of Bayesian networks using constraint programming. In *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 429–445. Springer, 2015.

[39] C. J. Verzilli, N. Stallard, and J. C. Whittaker. Bayesian graphical models for genome-wide association studies. *The American Journal of Human Genetics*, 79(1):100–112, 2006.

[40] A. Wiesel, Y. C. Eldar, and A. O. H. III. Covariance estimation in decomposable Gaussian graphical models. *IEEE Transactions on Signal Processing*, 58(3):1482–1492, 2010.

[41] C. Yuan and B. M. Malone. Learning optimal Bayesian networks: A shortest path perspective. *Journal of Artificial Intelligence Research*, 48:23–65, 2013.